IFW

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Lefebvre et al. Confirmation No. 3723

Application No.: 10/771,570 Group No.: 3621

Filed: February 4, 2004 Examiner: Not Yet Known

For: SYSTEM AND METHOD FOR ASSIGNING CREDIT TO PROCESS INPUTS

OK PR

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

THIRD-PARTY SUBMISSION IN PUBLISHED APPLICATION
(37 C.F.R. § 1.99)

1. The above identified application has been published on August 4, 2005.

   This submission is being made within two months from the date of publication of the above application.

2. Attached hereto is:

   A copy of each listed patent or publication in written form.

   1) U.S. Patent No. 4,208,712, filed November 2, 1970, issued June 17, 1980.

   2) U.S. Patent No. 5,353,207, filed June 10, 1992, issued October 4, 1994.

CERTIFICATION UNDER 37 C.F.R. §§ 1.8(a) and 1.10*

*(When using Express Mail, the Express Mail label number is **mandatory**;*
Express Mail certification is optional.)

I hereby certify that, on the date shown below, this correspondence is being:

MAILING

XX deposited with the United States Postal Service in an envelope addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

37 C.F.R. § 1.8(a)

XX with sufficient postage as first class mail.

37 C.F.R. § 1.10*

__ as "Express Mail Post Office to Addressee"
Mailing Label No. _____ **(mandatory)**

TRANSMISSION

__ facsimile transmitted to the Patent and Trademark Office, (703) _____ - _____.

10/07/2005 MBELETE1 00000001 10771570
01 FC:1806 180.00 OP

Laura K. Cahill
Signature

Date: October 3, 2005

Laura K. Cahill
(type or print name of person certifying)

** Only the date of filing (' 1.6) will be the date used in a patent term adjustment calculation, although the date on any certificate of mailing or transmission under ' 1.8 continues to be taken into account in determining timeliness. See ' 1.703(f). Consider "Express Mail Post Office to Addressee" (' 1.10) or facsimile transmission (' 1.6(d)) for the reply to be accorded the earliest possible filing date for patent term adjustment calculations.*

~~10/07/2005 MBELETE1 00000001 10771570~~

3) U.S. Patent No. 5,825,646, filed June 3, 1996, issued October 20, 1998.

4) U.S. Patent No. 6,278,899, filed October 6, 1998, issued August 21, 2001.

5) U.S. Patent No. 6,381,504, filed December 31, 1998, issued April 30, 2002.

6) U.S. Patent No. 6,678,585, filed September 28, 2000, issued January 13, 2004.

7) Bishop, C., "Neural Networks for Pattern Recognition," Oxford University Press, 1995, pp 140-150.

8) Piche, S., "Steepest Descent Algorithms for Neural Network Controllers and Filters," IEEE Transactions on Neural Networks, Vol. 5, No. 2, March 1994, pp 198-212.

3. Service of this paper and the attachments thereto have been made on the applicant and the proof of such service is attached.

4. Fees Due

Fee set forth in § 1.17(p): $180.00

Total fee(s) due $180.00

5. Payment of fees

Enclosed is Check No. 6602 in the amount of $180.00.

A duplicate of this paper is attached.

Date: **October 3, 2005**

Reg. No.: 36,326
Tel. No.: 440-684-1090

Customer No.: 22203

Signature of Practitioner
Michael A. Jaffe

Kusner & Jaffe
Highland Place - Suite 310
6151 Wilson Mills Road
Highland Heights, OH 44143

RICHMOND HTS. POSTAL STORE
RICHMOND HEIGHTS, Ohio
441439998
3816630431-0095
09/30/2005 (800)275-8777 04:51:49 PM

Sales Receipt

Product Description	Sale Qty	Unit Price	Final Price
BOSTON MA 02109 Priority Mail			$6.05
Return Receipt (Green Card)			$1.75
Certified			$2.30
Label Serial #: 70022410000417304078			
Customer Postage			-$10.10
Subtotal:			$0.00
Total:			$0.00

Paid by:

Bill#: 1000503140666
Clerk: 04

— All sales final on stamps and postage. —
Refunds for guaranteed services only.
Thank you for your business.
Customer Copy

7002 2410 0004 1720 4028

U.S. Postal Service™
CERTIFIED MAIL™ RECEIPT
(Domestic Mail Only; No Insurance Coverage Provided)

For delivery information visit our website at www.usps.com®

OFFICIAL USE

Postage	$ 6.05
Certified Fee	2.30
Return Reciept Fee (Endorsement Required)	1.75
Restricted Delivery Fee (Endorsement Required)	
Total Postage & Fees	$10.10

RICHMOND HEIGHTS BR. CLEVELAND, OH 44143
Postmark Here
SEP 30 2005
USPS

Sent To: Wilmer Cutler Pickering Hale and Dorr
Street, Apt. No., or PO Box No.: 60 State Street
City, State, ZIP+4: Boston, MA 02109

PS Form 3800, June 2002 See Reverse for Instructions

SENDER: COMPLETE THIS SECTION

- Complete items 1, 2, and 3. Also complete item 4 if Restricted Delivery is desired.
- Print your name and address on the reverse so that we can return the card to you.
- Attach this card to the back of the mailpiece, or on the front if space permits.

1. Article Addressed to:

Wilmer Cutler Pickering Hale
and Dorr LLP
60 State Street
Boston, MA 02109

COMPLETE THIS SECTION ON DELIVERY

A. Signature

X ☐ Agent ☐ Addressee

B. Received by (Printed Name) C. Date of Delivery

D. Is delivery address different from item 1? ☐ Yes
If YES, enter delivery address below: ☐ No

3. Service Type
☒ Certified Mail ☐ Express Mail
☐ Registered ☐ Return Receipt for Merchandise
☐ Insured Mail ☐ C.O.D.

4. Restricted Delivery? (Extra Fee) ☐ Yes

2. Article Number
(Transfer from service label) 7002 2410 0004 1730 4078

PS Form 3811, August 2001 Domestic Return Receipt 102595-02-M-1035.

UNITED STATES POSTAL SERVICE

First-Class Mail
Postage & Fees Paid
USPS
Permit No. G-10

• Sender: Please print your name, address, and ZIP+4 in this box •

KUSNER & JAFFE
HIGHLAND PLACE - SUITE 310
6151 WILSON MILLS ROAD
HIGHLAND HEIGHTS, OH 44143

7002 2410 0004 1730 4078

PLACE STICKER AT TOP OF ENVELOPE TO THE RIGHT OF THE RETURN ADDRESS. FOLD AT DOTTED LINE

CERTIFIED MAIL™

7002 2410 0004 1730 4078

7002 2410 0004 1730 4078

U.S. Postal Service™
CERTIFIED MAIL™ RECEIPT
(Domestic Mail Only; No Insurance Coverage Provided)

For delivery information visit our website at www.usps.com®

OFFICIAL USE

Postage	$
Certified Fee	
Return Reciept Fee (Endorsement Required)	
Restricted Delivery Fee (Endorsement Required)	
Total Postage & Fees	$

Postmark Here

Sent To: Wilmer Cutler Pickering Hale and Dorr

Street, Apt. No., or PO Box No.: 60 State Street

City, State, ZIP+4: Boston, MA 02109

PS Form 3800, June 2002 — See Reverse for Instructions

KUSNER & JAFFE

HIGHLAND PLACE • SUITE 310
6151 WILSON MILLS ROAD
HIGHLAND HEIGHTS, OHIO 44143

MARK KUSNER
MICHAEL A. JAFFE
THOMAS D. MCCLURE, JR.

TELEPHONE • (440) 684-1090
FACSIMILE • (440) 684-1095
E-MAIL • MAIL@KUSNERJAFFE.COM

INTELLECTUAL PROPERTY LAW
PATENTS • TRADEMARKS • COPYRIGHTS

September 30, 2005

CERTIFIED MAIL
NO. 7002 2410 0004 1730 4078

Wilmer Cutler Pickering Hale and Dorr LLP
60 State Street
Boston, MA 02109

Re: U.S. Application Serial No. 10/771,570
Published as Patent Application Publication No. US2005/0171880
Title: SYSTEM AND METHOD FOR ASSIGNING CREDIT TO PROCESS INPUTS
Attorney Docket No.: 113179-127

Dear Sir or Madam:

Pursuant to 37 C.F.R. 1.99 and 37 C.F.R. 1.248, enclosed please find a "Third Party Submission in a Published Application" being filed with the U.S. Patent and Trademark Office in connection with the above-identified patent application.

Sincerely yours,

Michael A. Jaffe, Esq.

MAJ/lc
Enclosures

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Lefebvre et al. Confirmation No. 3723

Application No.: 10/771,570 Group No.: 3621

Filed: February 4, 2004 Examiner: Not Yet Known

For: SYSTEM AND METHOD FOR ASSIGNING CREDIT TO PROCESS INPUTS

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

THIRD-PARTY SUBMISSION IN PUBLISHED APPLICATION (37 C.F.R. § 1.99)

1. The above identified application has been published on August 4, 2005.

   This submission is being made within two months from the date of publication of the above application.

2. Attached hereto is:

   A copy of each listed patent or publication in written form.

   1) U.S. Patent No. 4,208,712, filed November 2, 1970, issued June 17, 1980.

   2) U.S. Patent No. 5,353,207, filed June 10, 1992, issued October 4, 1994.

CERTIFICATION UNDER 37 C.F.R. §§ 1.8(a) and 1.10*

*(When using Express Mail, the Express Mail label number is **mandatory**; Express Mail certification is optional.)*

I hereby certify that, on the date shown below, this correspondence is being:

MAILING

XX deposited with the United States Postal Service in an envelope addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

37 C.F.R. § 1.8(a)

XX with sufficient postage as first class mail.

37 C.F.R. § 1.10*

__ as "Express Mail Post Office to Addressee"
Mailing Label No. _____ **(mandatory)**

TRANSMISSION

__ facsimile transmitted to the Patent and Trademark Office, (703) _____ - _____.

_____
Signature

Date: _____

_____
(type or print name of person certifying)

** Only the date of filing (' 1.6) will be the date used in a patent term adjustment calculation, although the date on any certificate of mailing or transmission under ' 1.8 continues to be taken into account in determining timeliness. See ' 1.703(f). Consider "Express Mail Post Office to Addressee" (' 1.10) or facsimile transmission (' 1.6(d)) for the reply to be accorded the earliest possible filing date for patent term adjustment calculations.*

3) U.S. Patent No. 5,825,646, filed June 3, 1996, issued October 20, 1998.

4) U.S. Patent No. 6,278,899, filed October 6, 1998, issued August 21, 2001.

5) U.S. Patent No. 6,381,504, filed December 31, 1998, issued April 30, 2002.

6) U.S. Patent No. 6,678,585, filed September 28, 2000, issued January 13, 2004.

7) Bishop, C., "Neural Networks for Pattern Recognition," Oxford University Press, 1995, pp 140-150.

8) Piche, S., "Steepest Descent Algorithms for Neural Network Controllers and Filters," IEEE Transactions on Neural Networks, Vol. 5, No. 2, March 1994, pp 198-212.

3. Service of this paper and the attachments thereto have been made on the applicant and the proof of such service is attached.

4. Fees Due

Fee set forth in § 1.17(p): $180.00

Total fee(s) due $180.00

5. Payment of fees

Enclosed is Check No. 6602 in the amount of $180.00.

A duplicate of this paper is attached.

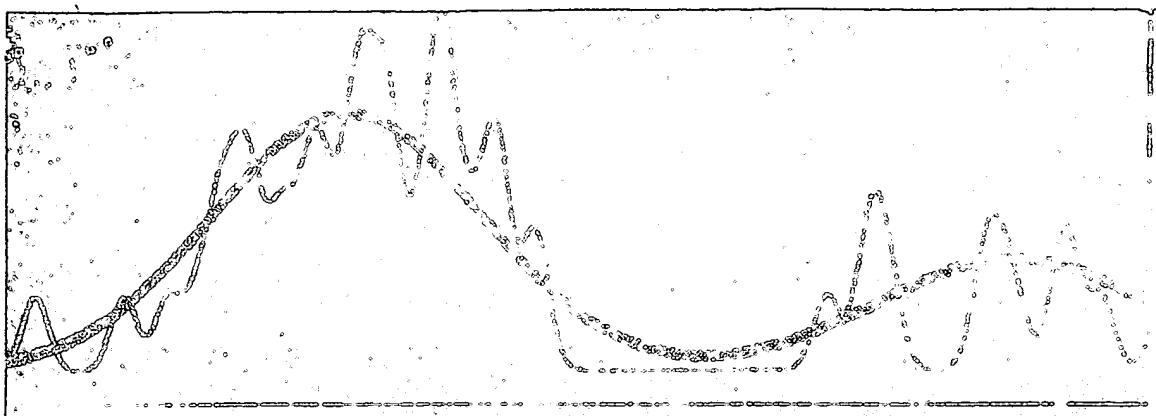
Date: _____

Reg. No.: 36,326
Tel. No.: 440-684-1090

Customer No.: 22203

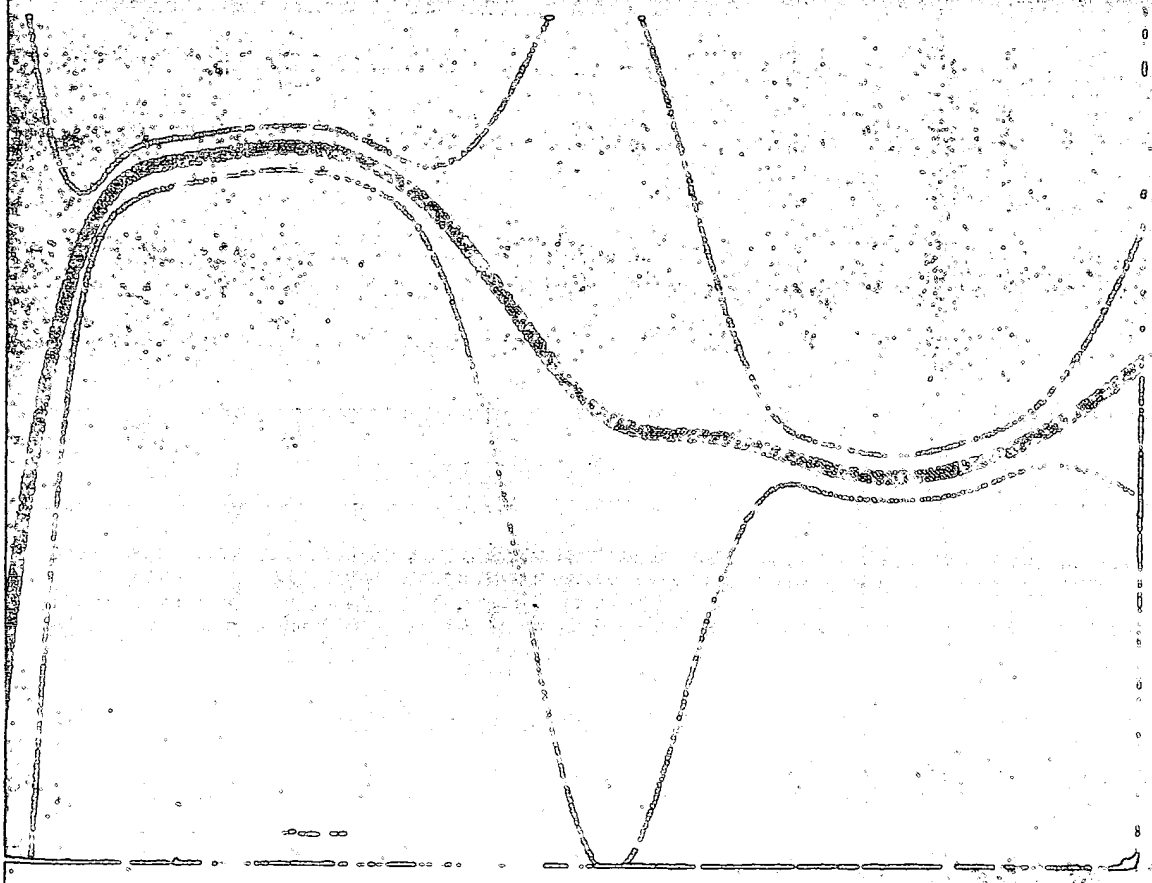
_____
Signature of Practitioner
Michael A. Jaffe

Kusner & Jaffe
Highland Place - Suite 310
6151 Wilson Mills Road
Highland Heights, OH 44143



Neural Networks for Pattern Recognition

Christopher M. Bishop



Oxford University Press, Walton Street, Oxford OX2 6DP
Oxford New York
Athens Auckland Bangkok Bombay
Calcutta Cape Town Dar es Salaam Delhi
Florence Hong Kong Istanbul Karachi
Kuala Lumpur Madras Madrid Melbourne
Mexico City Nairobi Paris Singapore
Taipei Tokyo Toronto
and associated companies in
Berlin Ibadan

Oxford is a trade mark of Oxford University Press

Published in the United States by
Oxford University Press Inc., New York

First published 1995
Reprinted 1996 (twice)

A catalogue record for this book is available from the British Library

Library of Congress Cataloging in Publication Data
Bishop, Chris (Chris M.)
Neural networks for pattern recognition / Chris Bishop.
1. Neural networks (Computer science). 2. Pattern recognition systems. I. Title.
QA76.87.B574 1995 006.4—dc20 95-40465
ISBN 0 19 853849 9 (Hbk)
ISBN 0 19 853864 2 (Pbk)

Printed in Great Britain by
Bookcraft Ltd, Midsomer Norton, Avon

to the input variables. Smoothness of the network mapping is an important property in connection with the generalization performance of a network, as is discussed in greater detail in Section 9.2. The second reason is that the function g depends on the particular function $y(\mathbf{x})$ which we wish to represent. This is the converse of the situation which we generally encounter with neural networks. Usually, we consider fixed activation functions, and then adjust the number of hidden units, and the values of the weights and biases, to give a sufficiently close representation of the desired mapping. In Kolmogorov's theorem the number of hidden units is fixed, while the activation functions depend on the mapping. In general, if we are trying to represent an arbitrary continuous function then we cannot hope to do this exactly with a finite number of fixed activation functions since the finite number of adjustable parameters represents a finite number of degrees of freedom, and a general continuous function has effectively infinitely many degrees of freedom.

4.8 Error back-propagation

So far in this chapter we have concentrated on the representational capabilities of multi-layer networks. We next consider how such a network can learn a suitable mapping from a given data set. As in previous chapters, learning will be based on the definition of a suitable error function, which is then minimized with respect to the weights and biases in the network.

Consider first the case of networks of threshold units. The final layer of weights in the network can be regarded as a perceptron with inputs given by the outputs of the last layer of hidden units. These weights could therefore be chosen using the perceptron learning rule introduced in Chapter 3. Such an approach cannot, however, be used to determine the weights in earlier layers of the network. Although such layers could in principle be regarded as being like single-layer perceptrons, we have no procedure for assigning target values to their outputs, and so the perceptron procedure cannot be applied. This is known as the *credit assignment problem*. If an output unit produces an incorrect response when the network is presented with an input vector we have no way of determining which of the hidden units should be regarded as responsible for generating the error, so there is no way of determining which weights to adjust or by how much.

The solution to this credit assignment problem is relatively simple. If we consider a network with differentiable activation functions, then the activations of the output units become differentiable functions of both the input variables, and of the weights and biases. If we define an error function, such as the sum-of-squares error introduced in Chapter 1, which is a differentiable function of the network outputs, then this error is itself a differentiable function of the weights. We can therefore evaluate the derivatives of the error with respect to the weights, and these derivatives can then be used to find weight values which minimize the error function, by using either gradient descent or one of the more powerful optimization methods discussed in Chapter 7. The algorithm for evaluating the derivatives of the error function is known as *back-propagation* since, as we shall

see, it corresponds to a propagation of errors backwards through the network. The technique of back-propagation was popularized in a paper by Rumelhart, Hinton and Williams (1986). However, similar ideas had been developed earlier by a number of researchers including Werbos (1974) and Parker (1985).

It should be noted that the term back-propagation is used in the neural computing literature to mean a variety of different things. For instance, the multi-layer perceptron architecture is sometimes called a back-propagation network. The term back-propagation is also used to describe the training of a multi-layer perceptron using gradient descent applied to a sum-of-squares error function. In order to clarify the terminology it is useful to consider the nature of the training process more carefully. Most training algorithms involve an iterative procedure for minimization of an error function, with adjustments to the weights being made in a sequence of steps. At each such step we can distinguish between two distinct stages. In the first stage, the derivatives of the error function with respect to the weights must be evaluated. As we shall see, the important contribution of the back-propagation technique is in providing a computationally efficient method for evaluating such derivatives. Since it is at this stage that errors are propagated backwards through the network, we shall use the term back-propagation specifically to describe the evaluation of derivatives. In the second stage, the derivatives are then used to compute the adjustments to be made to the weights. The simplest such technique, and the one originally considered by Rumelhart *et al.* (1986), involves gradient descent. It is important to recognize that the two stages are distinct. Thus, the first stage process, namely the propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multi-layer perceptron. It can also be applied to error functions other that just the simple sum-of-squares, and to the evaluation of other derivatives such as the Jacobian and Hessian matrices, as we shall see later in this chapter. Similarly, the second stage of weight adjustment using the calculated derivatives can be tackled using a variety of optimization schemes (discussed at length in Chapter 7), many of which are substantially more powerful than simple gradient descent.

4.8.1 *Evaluation of error function derivatives*

We now derive the back-propagation algorithm for a general network having arbitrary feed-forward topology, and arbitrary differentiable non-linear activation functions, for the case of an arbitrary differentiable error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units and a sum-of-squares error.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{4.26}$$

where z_i is the activation of a unit, or input, which sends a connection to unit

j, and w_{ji} is the weight associated with that connection. The summation runs over all units which send connections to unit j. In Section 4.1 we showed that biases can be included in this sum by introducing an extra unit, or input, with activation fixed at +1. We therefore do not need to deal with biases explicitly. The sum in (4.26) is transformed by a non-linear activation function $g(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = g(a_j). \tag{4.27}$$

Note that one or more of the variables z_i in the sum in (4.26) could be an input, in which case we shall denote it by x_i. Similarly, the unit j in (4.27) could be an output unit, in which case we denote its activation by y_k.

As before, we shall seek to determine suitable values for the weights in the network by minimization of an appropriate error function. Here we shall consider error functions which can be written as a sum, over all patterns in the training set, of an error defined for each pattern separately

$$E = \sum_n E^n \tag{4.28}$$

where n labels the patterns. Nearly all error functions of practical interest take this form, for reasons which are explained in Chapter 6. We shall also suppose that the error E^n can be expressed as a differentiable function of the network output variables so that

$$E^n = E^n(y_1, \ldots, y_c). \tag{4.29}$$

Our goal is to find a procedure for evaluating the derivatives of the error function E with respect to the weights and biases in the network. Using (4.28) we can express these derivatives as sums over the training set patterns of the derivatives for each pattern separately. From now on we shall therefore consider one pattern at a time.

For each pattern we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (4.26) and (4.27). This process is often called *forward propagation* since it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E^n with respect to some weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. However, in order to keep the notation uncluttered, we shall omit the superscript n from the input and activation variables. First we note that E^n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{4.30}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} \tag{4.31}$$

where the δ's are often referred to as *errors* for reasons we shall see shortly. Using (4.26) we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{4.32}$$

Substituting (4.31) and (4.32) into (4.30) we then obtain

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_i. \tag{4.33}$$

Note that this has the same general form as obtained for single-layer networks in Section 3.4. Equation (4.33) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (4.33).

For the output units the evaluation of δ_k is straightforward. From the definition (4.31) we have

$$\delta_k \equiv \frac{\partial E^n}{\partial a_k} = g'(a_k)\frac{\partial E^n}{\partial y_k} \tag{4.34}$$

where we have used (4.27) with z_k denoted by y_k. In order to evaluate (4.34) we substitute appropriate expressions for $g'(a)$ and $\partial E^n/\partial y$. This will be illustrated with a simple example shortly.

To evaluate the δ's for hidden units we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{4.35}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 4.16. Note that the units labelled k could include other hidden units and/or output units. In writing

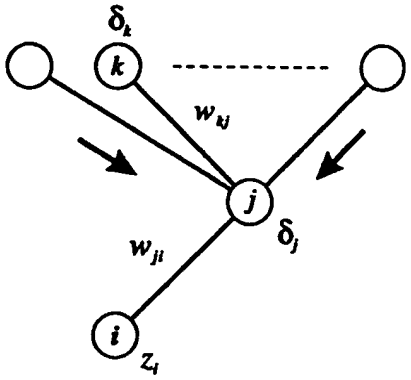


Figure 4.16. Illustration of the calculation of δ_j for hidden unit j by back-propagation of the δ's from those units k to which unit j sends connections.

down (4.35) we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (4.31) into (4.35), and make use of (4.26) and (4.27), we obtain the following *back-propagation* formula

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k \tag{4.36}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in Figure 4.16. Since we already know the values of the δ's for the output units, it follows that by recursively applying (4.36) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology.

We can summarize the back-propagation procedure for evaluating the derivatives of the error E^n with respect to the weights in four steps:

1. Apply an input vector $\mathbf{x}^n$ to the network and forward propagate through the network using (4.26) and (4.27) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (4.34).
3. Back-propagate the δ's using (4.36) to obtain δ_j for each hidden unit in the network.
4. Use (4.33) to evaluate the required derivatives.

The derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set, and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}}. \tag{4.37}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $g(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $g(\cdot)$ goes with which unit.

4.8.2 *A simple example*

The above derivation of the back-propagation procedure allowed for general forms for the error function, the activation functions and the network topology. In order to illustrate the application of this algorithm, we shall consider a particular example. This is chosen both for its simplicity and for its practical importance, since many applications of neural networks reported in the literature make use of this type of network. Specifically, we shall consider a two-layer network of the form illustrated in Figure 4.1, together with a sum-of-squares error. The output units have linear activation functions while the hidden units have logistic sigmoid activation functions given by (4.10), and repeated here:

$$g(a) \equiv \frac{1}{1 + \exp(-a)}. \tag{4.38}$$

A useful feature of this function is that its derivative can be expressed in a particularly simple form:

$$g'(a) = g(a)(1 - g(a)). \tag{4.39}$$

In a software implementation of the network algorithm, (4.39) represents a convenient property since the derivative of the activation can be obtained efficiently from the activation itself using two arithmetic operations.

For the standard sum-of-squares error function, the error for pattern n is given by

$$E^n = \frac{1}{2}\sum_{k=1}^{c}(y_k - t_k)^2 \tag{4.40}$$

where y_k is the response of output unit k, and t_k is the corresponding target, for a particular input pattern $\mathbf{x}^n$.

Using the expressions derived above for back-propagation in a general network, together with (4.39) and (4.40), we obtain the following results. For the output units, the δ's are given by

$$\delta_k = y_k - t_k \tag{4.41}$$

while for units in the hidden layer the δ's are found using

$$\delta_j = z_j(1 - z_j)\sum_{k=1}^{c} w_{kj}\delta_k \tag{4.4}$$

where the sum runs over all output units. The derivatives with respect to tl first-layer and second-layer weights are then given by

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j x_i, \qquad \frac{\partial E^n}{\partial w_{kj}} = \delta_k z_j. \tag{4.4}$$

So far we have discussed the evaluation of the derivatives of the error functio with respect to the weights and biases in the network. In order to turn this int a learning algorithm we need some method for updating the weights based o these derivatives. In Chapter 7 we discuss several such parameter optimizatio strategies in some detail. For the moment, we consider the fixed-step gradien descent technique introduced in Section 3.4. We have the choice of updating th weights either after presentation of each pattern (on-line learning) or after firs summing the derivatives over all the patterns in the training set (batch learning) In the former case the weights in the first layer are updated using

$$\Delta w_{ji} = -\eta\delta_j x_i \tag{4.44}$$

while in the case of batch learning the first-layer weights are updated using

$$\Delta w_{ji} = -\eta\sum_n \delta_j^n x_i^n \tag{4.45}$$

with analogous expressions for the second-layer weights.

4.8.3 *Efficiency of back-propagation*

One of the most important aspects of back-propagation is its computational efficiency. To understand this, let us examine how the number of computer operations required to evaluate the derivatives of the error function scales with the size of the network. Let W be the total number of weights and biases. Then a single evaluation of the error function (for a given input pattern) would require $\mathcal{O}(W)$ operations, for sufficiently large W. This follows from the fact that, except for a network with very sparse connections, the number of weights is typically much greater than the number of units. Thus, the bulk of the computational effort in forward propagation is concerned with evaluating the sums in (4.26), with the evaluation of the activation functions representing a small overhead. Each term in the sum in (4.26) requires one multiplication and one addition, leading to an overall computational cost which is $\mathcal{O}(W)$.

For W weights in total there are W such derivatives to evaluate. If we simply

took the expression for the error function and wrote down explicit formulae for the derivatives and then evaluated them numerically by forward propagation, we would have to evaluate W such terms (one for each weight or bias) each requiring $\mathcal{O}(W)$ operations. Thus, the total computational effort required to evaluate all the derivatives would scale as $\mathcal{O}(W^2)$. By comparison, back-propagation allows the derivatives to be evaluated in $\mathcal{O}(W)$ operations. This follows from the fact that both the forward and the backward propagation phases are $\mathcal{O}(W)$, and the evaluation of the derivative using (4.33) also requires $\mathcal{O}(W)$ operations. Thus back-propagation has reduced the computational complexity from $\mathcal{O}(W^2)$ to $\mathcal{O}(W)$ for each input vector. Since the training of MLP networks, even using back-propagation, can be very time consuming, this gain in efficiency is crucial. For a total of N training patterns, the number of computational steps required to evaluate the complete error function for the whole data set is N times larger than for one pattern.

The practical importance of the $\mathcal{O}(W)$ scaling of back-propagation is analogous in some respects to that of the fast Fourier transform (FFT) algorithm (Brigham, 1974; Press *et al.*, 1992) which reduces the computational complexity of evaluating an L-point Fourier transform from $\mathcal{O}(L^2)$ to $\mathcal{O}(L \log_2 L)$. The discovery of this algorithm led to the widespread use of Fourier transforms in a large range of practical applications.

4.8.4 *Numerical differentiation*

An alternative approach to back-propagation for computing the derivatives of the error function is to use finite differences. This can be done by perturbing each weight in turn, and approximating the derivatives by the expression

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{E^n(w_{ji} + \epsilon) - E^n(w_{ji})}{\epsilon} + \mathcal{O}(\epsilon) \tag{4.46}$$

where $\epsilon \ll 1$ is a small quantity. In a software simulation, the accuracy of the approximation to the derivatives can be improved by making ϵ smaller, until numerical roundoff problems arise. The main problem with this approach is that the highly desirable $\mathcal{O}(W)$ scaling has been lost. Each forward propagation requires $\mathcal{O}(W)$ steps, and there are W weights in the network each of which must be perturbed individually, so that the overall scaling is $\mathcal{O}(W^2)$. However, finite differences play an important role in practice, since a numerical comparison of the derivatives calculated by back-propagation with those obtained using finite differences provides a very powerful check on the correctness of any software implementation of the back-propagation algorithm.

The accuracy of the finite differences method can be improved significantly by using symmetrical ***central differences*** of the form

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{E^n(w_{ji} + \epsilon) - E^n(w_{ji} - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{4.47}$$

In this case the $\mathcal{O}(\epsilon)$ corrections cancel, as is easily verified by Taylor expansion on the right-hand side of (4.47), and so the residual corrections are $\mathcal{O}(\epsilon^2)$. The number of computational steps is, however, roughly doubled compared with (4.46).

We have seen that the derivatives of an error function with respect to the weights in a network can be expressed efficiently through the relation

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} z_i. \tag{4.48}$$

Instead of using the technique of central differences to evaluate the derivatives $\partial E^n/\partial w_{ji}$ directly, we can use it to estimate $\partial E^n/\partial a_j$ since

$$\frac{\partial E^n}{\partial a_j} = \frac{E^n(a_j + \epsilon) - E^n(a_j - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2) \tag{4.49}$$

We can then make use of (4.48) to evaluate the required derivatives. Because the derivatives with respect to the weights are found from (4.48) this approach is still relatively efficient. Back-propagation requires one forward and one backward propagation through the network, each taking $\mathcal{O}(W)$ steps, in order to evaluate all of the $\partial E/\partial a_i$. By comparison, (4.49) requires $2M$ forward propagations, where M is the number of hidden and output nodes. The overall scaling is therefore proportional to MW, which is typically much less than the $\mathcal{O}(W^2)$ scaling of (4.47), but more than the $\mathcal{O}(W)$ scaling of back-propagation. This technique is called *node perturbation* (Jabri and Flower, 1991), and is closely related to the *madeline III* learning rule (Widrow and Lehr, 1990).

In a software implementation, derivatives should be evaluated using back-propagation, since this gives the greatest accuracy and numerical efficiency. However, the results should be compared with numerical differentiation using (4.47) for a few test cases in order to check the correctness of the implementation.

4.9 The Jacobian matrix

We have seen how the derivatives of an error function with respect to the weights can be obtained by the propagation of errors backwards through the network. The technique of back-propagation can also be applied to the calculation of other derivatives. Here we consider the evaluation of the *Jacobian* matrix, whose elements are given by the derivatives of the network outputs with respect to the inputs

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i} \tag{4.50}$$

where each such derivative is evaluated with all other inputs held fixed. Note that the term Jacobian matrix is also sometimes used to describe the derivatives

of the error function with respect to the network weights, as calculated earlier using back-propagation. The Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables, and is useful in several contexts in the application of neural networks. For instance, if there are known errors associated with the input variables, then the Jacobian matrix allows these to be propagated through the trained network in order to estimate their contribution to the errors at the outputs. Thus, we have

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i. \tag{4.51}$$

In general, the network mapping represented by a trained neural network will be non-linear, and so the elements of the Jacobian matrix will not be constants but will depend on the particular input vector used. Thus (4.51) is valid only for small perturbations of the inputs, and the Jacobian itself must be re-evaluated for each new input vector.

The Jacobian matrix can be evaluated using a back-propagation procedure which is very similar to the one derived earlier for evaluating the derivatives of an error function with respect to the weights. We start by writing the element J_{ki} in the form

$$\begin{aligned} J_{ki} = \frac{\partial y_k}{\partial x_i} &= \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} \\ &= \sum_j w_{ji} \frac{\partial y_k}{\partial a_j} \end{aligned} \tag{4.52}$$

where we have made use of (4.26). The sum in (4.52) runs over all units j to which the input unit i sends connections (for example, over all units in the first hidden layer in the layered topology considered earlier). We now write down a recursive back-propagation formula to determine the derivatives $\partial y_k / \partial a_j$

$$\begin{aligned} \frac{\partial y_k}{\partial a_j} &= \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} \\ &= g'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l} \end{aligned} \tag{4.53}$$

where the sum runs over all units l to which unit j sends connections. Again, we have made use of (4.26) and (4.27). This back-propagation starts at the output units for which, using (4.27), we have

$$\frac{\partial y_k}{\partial a_{k'}} = g'(a_k)\delta_{kk'} \tag{4.54}$$

where $\delta_{kk'}$ is the Kronecker delta symbol, and equals 1 if $k = k'$ and 0 otherwise. We can therefore summarize the procedure for evaluating the Jacobian matrix as follows. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be found, and forward propagate in the usual way to obtain the activations of all of the hidden and output units in the network. Next, for each row k of the Jacobian matrix, corresponding to the output unit k, back-propagate using the recursive relation (4.53), starting with (4.54), for all of the hidden units in the network. Finally, use (4.52) to do the back-propagation to the inputs. The second and third steps are then repeated for each value of k, corresponding to each row of the Jacobian matrix.

The Jacobian can also be evaluated using an alternative *forward* propagation formalism which can be derived in an analogous way to the back-propagation approach given here (Exercise 4.6). Again, the implementation of such algorithms can be checked by using numerical differentiation in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{4.55}$$

4.10 The Hessian matrix

We have shown how the technique of back-propagation can be used to obtain the first derivatives of an error function with respect to the weights in the network. Back-propagation can also be used to evaluate the second derivatives of the error, given by

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}}. \tag{4.56}$$

These derivatives form the elements of the *Hessian* matrix, which plays an important role in many aspects of neural computing, including the following:

1. Several non-linear optimization algorithms used for training neural networks are based on considerations of the second-order properties of the error surface, which are controlled by the Hessian matrix (Chapter 7).
2. The Hessian forms the basis of a fast procedure for re-training a feed-forward network following a small change in the training data (Bishop, 1991a).
3. The inverse of the Hessian has been used to identify the least significant weights in a network as part of network 'pruning' algorithms (Section 9.5.3).
4. The inverse of the Hessian can also be used to assign error bars to the predictions made by a trained network (Section 10.2).

Steepest Descent Algorithms for Neural Network Controllers and Filters

Stephen W. Piché, *Member, IEEE*

Abstract—**A number of steepest descent algorithms have been developed for adapting discrete-time dynamical systems, including the backpropagation through time and recursive backpropagation algorithms. In this paper, a tutorial on the use of these algorithms for adapting neural network controllers and filters is presented. In order to effectively compare and contrast the algorithms, a unified framework for the algorithms is developed. This framework is based upon a standard representation of a discrete-time dynamical system. Using this framework, the computational and storage requirements of the algorithms are derived. These requirements are used to select the appropriate algorithm for training a neural network controller or filter. Finally, to illustrate the usefulness of the techniques presented in this paper, a neural network control example and a neural network filtering example are presented.**

I. Introduction

THE ABILITY of humans to control and interact with a complex environment has motivated our study of adaptive discrete-time dynamical systems which are fully or at least partially composed of neural networks. Humans regularly perform certain tasks easily and proficiently which have proven difficult to reproduce with machines. Examples of such tasks include driving a car, recognizing the differences between a cat and a dog, and understanding spoken language. Humans learn how to accomplish these tasks in part by interacting with, manipulating and eventually controlling their environment. In order to build machines which accomplish these difficult tasks, it may be necessary both to mimic humans learning through environmental interaction and to model the low level functions of the human nervous system. The algorithms presented in this paper provide one possible technique for accommodating both of these requirements.

Although the human ability to control its environment has motivated us, our primary interest lies in the development of engineering tools for adaptive nonlinear control and filtering. Specifically, we have investigated adaptive algorithms that allow the design of closed-loop nonlinear controllers and nonlinear infinite impulse response (IIR) filters. Sigmoidal neural networks of the type described by Rumelhart [1] are used to implement the controllers and filters. Neural networks of this type were selected because they are capable of learning a nonlinear function to any arbitrary degree of accuracy [2], [3].

A. A Unified Framework

The algorithms presented in this paper are based on steepest descent methods [4], [5] for calculating changes to adaptive parameters of a discrete-time dynamical system. Most of the algorithms presented in this paper have been previously published. Our goal in this paper is to present these algorithms in a unified framework so that their application to neural network control and filtering can be easily understood.

It is our hope that this unified framework will prove useful to those who are both familiar and unfamiliar with steepest descent approaches to training discrete-time dynamical systems. For those who are not familiar with these algorithms, we hope that the unified treatment of the theory makes it easier to understand the major differences between the various algorithms. For those who are familiar with these algorithms, we hope that the unified framework provides insight into some of the more subtle differences.

In this paper, we divide the theory concerning the training of adaptive discrete-time dynamical systems into three distinct layers. The three layers include the mathematical theory, general techniques for training dynamical systems, and specific algorithms for training dynamical systems composed of neural networks. The three layers are hierarchically structured with the mathematical theory at the top and the specific algorithms at the bottom. As shown in this paper, by starting at the top layer and working down, a natural unified framework for the steepest descent algorithms emerges.

The top layer includes the mathematical theory concerning the calculation of an error gradient, which is used to drive the steepest descent algorithms. This theory, which was developed by Paul Werbos and is well-developed in his thesis [6], provides a formal technique for calculating gradients of discrete-time dynamical systems. As shown in Section III, his theory is based upon the ordered partial derivative [6].

The middle layer of the theory is composed of two general algorithms for updating the weights of an adaptive discrete-time dynamical system.[1] As shown in this paper, the two techniques, which are referred to as the *backsweep* and *recursive* algorithms, result from the use of different chain rule

Manuscript received July 5, 1991; revised February 1, 1993. This research was conducted at Stanford University and was funded by the following organizations: Electric Power Research Institute under contract #RP 8010-13, Dept. of the Army, Belvoir RD&E Center under contract #DAAK70-92-K-0003, Office of Naval Research under contract #N000014-92-J-1787, and the National Science Foundation under contract #IRI 9113492-A1.

The author is with Microelectronic and Computer Technology Corporation (MCC), Austin, TX 78759.

IEEE Log Number 9214798.

[1] As shown in Section V, the general algorithms must be divided into epochwise or on-line versions (epochwise and on-line refer to the mode of operation of the dynamical system); therefore, there are actually four general algorithms. In this introduction section, it is convenient to ignore the epochwise and on-line difference, and treat the four general algorithms as if they were only two.

expansions of the ordered partial derivative. Both techniques have their origins in the 1960s. The backsweep technique for calculating the gradient of a discrete-time dynamical system with a single tap-delayed output was developed by Bryson [7]. This technique has been extensively applied in the field of optimal control. In this paper, we present the multiple-delayed version of this algorithm. The recursive technique for adapting dynamical systems was developed independently by Narendra and McBride [8] [9], and Kokotovic [10]. (A paper by Narendra and Parthasarathy [11] reviews the relationship between the work done in the 1960's and recent research on neural networks.) Because the recursive algorithm is well-suited for real-time learning, it's earliest applications were in the area of adaptive control.

The lowest layer of the theory includes the specific algorithms for training different types of dynamical systems. This layer includes the specific algorithms used to train neural network controllers and filters. An example of one of the algorithms which compose the lowest level of the theory is the backpropagation through time algorithm [1] [12]–[18]. This algorithm consists of using the general backsweep technique to train a specific architecture, a dynamical system composed solely of a feedforward neural network.[2] As shown in this paper, the backpropagation through time algorithm is characterized by low computational cost and high storage requirements. Furthermore, it is not an inherently on-line algorithm. The algorithm was popularized by its description in Chapter 8 of the well-known PDP book [1]. It was further popularized by its use for several control applications [12]–[14]. A number of good papers on backpropagation through time have been published over the past few years, including [19]–[28]. Most of the applications of this algorithm have been in the fields of control and speech processing. Because not all the details of the backpropagation through time algorithm are covered in this paper, a reader interested in further insights should refer to the referenced papers.

The use of the recursive technique for training dynamical systems composed of neural networks, which is referred to as the recursive backpropagation algorithm, is another example of an algorithm which is a member of the lowest level of the theory. As shown in this paper, the recursive backpropagation algorithm [15], [16], [29]–[32], which is often referred to as the real-time learning algorithm, is characterized by high computational cost and low storage requirements. Furthermore, it is an inherently on-line learning algorithm which is well suited for real-time learning. The algorithm was initially proposed by a number of different researchers [15], [16], [29]–[32]. Important applications of the algorithm have occurred in the areas of control [11], [33], [34], and speech processing [15], [16], [32], [35].

A review of many of the fundamental papers on the backpropagation through time and recursive backpropagation algorithms is given in [36]. Finally, in [37], the relationship between the dynamical neural network algorithms is shown using graph theory.

[2]It is important to note that the backsweep technique and backpropagation through time algorithm are not one and the same. Instead, backpropagation through time is the application of the backsweep technique to a specific dynamical system architecture, the feedforward neural network.

B. Potential Drawbacks of the Algorithms

Before proceeding with the development of the algorithms, two important drawbacks to training dynamical systems must be pointed out. First, for discrete-time dynamical systems trained using steepest descent algorithms, the weights (adaptive parameters) are not guaranteed to converge to values which would result in the global minimum of the error function. Neural network researchers should not find this surprising because training algorithms for static feedforward neural network have the same drawback. However, it should be noted that, in our experience, convergence to a non-acceptable local minimum is a more significant problem when training dynamical systems than static systems. Second, the stability of the weight updates cannot be guaranteed because analytic stability conditions for the training algorithms have not yet been developed. Stability is also a problem when training feedforward neural networks, but once again, in our experience, stability is a more significant problem when training dynamical networks.

These two drawbacks can be overcome by off-line development of the adaptive dynamical system. In such cases, if either non-optimal performance or instability occurs, the system can simply be retrained. If real-time applications of these algorithms are desired, conservative values of the training parameters should be selected to partially overcome these two problems. In such cases, the cost of non-optimal performance or instability should be small.

Finally, it should be noted that in some cases it is possible to short-circuit the algorithms presented in this paper by feeding back the desired output response instead of the actual output. In this case, the feedback loop is broken and simpler, static algorithms can be used to train the adaptive dynamical system. This method of training dynamical systems is referred to as teacher forcing [29], [38], [39] and generally is much more computationally efficient than the algorithms presented in this paper. Hence, if the desired output responses are available at each iteration, it is probably best to try teacher forcing methods before resorting to the algorithms present in this paper. However, for many control and filtering applications, such as the two given in Section VII of this paper, the desired output responses are not available at each iteration and the computationally more expensive dynamical algorithms must be used.

C. Outline

This paper is composed of eight sections and two appendices. Section II introduces a standard representation for a discrete-time dynamical system and illustrates how this representation can be used to model neural network controllers and filters. The mathematical theory concerning discrete-time dynamical systems is presented in Section III. Section IV presents the differences between epochwise and on-line training. The general gradient calculation techniques along with the specific algorithms for adapting dynamical neural networks

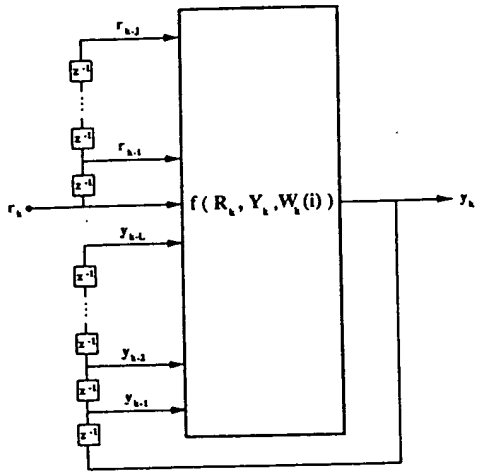


Fig. 1. Standard representation.

are given in Section V. Detailed derivations of the general algorithms are contained in the Appendices A and B. A comparison of the computational and storage requirements of the dynamical neural network algorithms is included in Section VI. This section also includes a discussion of the appropriate algorithms to use for neural network control and filtering applications. Section VII presents two applications of the algorithms, and Section VIII provides a conclusion.

II. Definitions

In this section, in order to provide a unified framework, a standard representation of a discrete-time dynamical system is proposed. It is shown that any discrete-time dynamical system, such as a neural network control system or IIR filter, can be expressed in this representation.

A. The Standard Representation

Fig. 1 shows a standard representation of a discrete-time dynamical system. This representation was introduced by Narendra [33]. In this figure, k denotes the iteration of the discrete-time dynamical system ($k = 0$ represents the first iteration). The input of the system is composed of two components. The first component, $\mathbf{R}_k$, is made up of an external input vector $\mathbf{r}_k$ and J delayed versions of this vector, $\mathbf{r}_{k-1}, \ldots, \mathbf{r}_{k-J}$. Each of the external input vectors is a length M column vector ($\mathbf{r}_k \in R^{[M\times 1]}$); therefore, $\mathbf{R}_k$ may be defined as a length $(J+1)M$ column vector of the form $\mathbf{R}_k = [\mathbf{r}_k^T, \mathbf{r}_{k-1}^T, \ldots, \mathbf{r}_{k-J}^T]^T$. The second component, $\mathbf{Y}_k$, is composed of the previous L output vectors. The output vector at iteration k is a length N column vector ($\mathbf{y}_k \in R^{[N\times 1]}$). Therefore, $\mathbf{Y}_k$ is a length LN column vector of the form $\mathbf{Y}_k = [\mathbf{y}_{k-1}^T, \mathbf{y}_{k-2}^T, \ldots, \mathbf{y}_{k-L}^T]^T$.

The weight vector $\mathbf{W}(i)$, which contains all adaptive parameters of the dynamical system, is defined as a length Q column vector ($\mathbf{W}(i) \in R^{[Q\times 1]}$). The index i denotes the number of times the weight vector has been updated using a steepest descent rule. The initial weight vector $\mathbf{W}(0)$ is usually produced by a random number generator. We use $\mathbf{W}_k(i)$ to denote the use of $\mathbf{W}(i)$ at iteration k of the dynamical system. Finally, let $w_k(i)$ denote any weight of the vector $\mathbf{W}_k(i)$.

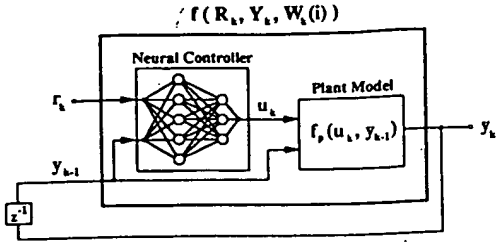


Fig. 2. neural network controller and plant in the standard representation.

By including *all* feedback states of a discrete-time dynamical system in the output vector $\mathbf{y}_k$, any dynamical system can be written as

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)), \tag{1}$$

where the function $f(\cdot)$ contains no delays [33]. The steepest descent algorithms of Section V are all defined in terms of this standard representation.

B. Standard Representations of Control Systems and Filters

In this section, in order to illustrate the flexibility of the standard representation, we present two discrete-time dynamical systems in their standard representations. Figure 2 shows how the standard representation may be used to model a neural network based control system. This dynamical system is composed of two basic components: a neural network controller and a plant model.[3] These two components are generically represented by $f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i))$ in the standard representation.

As shown in Fig. 2, the neural network control system operates in the following manner: An external reference signal, $\mathbf{r}_k$, is fed directly into a neural network controller. Given this reference signal along with the state of the plant model, which is assumed to be contained in $\mathbf{y}_{k-1}$, the neural network controller computes the control signal $\mathbf{u}_k$. The output of the plant model, $\mathbf{y}_k$, is calculated using the control signal, $\mathbf{u}_k$, and the state of the model, $\mathbf{y}_{k-1}$. It should be pointed out that it is assumed that all states of the discrete-time plant model which are contained in the plant's feedback loop are included in the output vector $\mathbf{y}_k$. Under this assumption, it can be observed that the neural network control system of Fig. 2 is given in its standard representation.

Before proceeding to the next example of a system in its standard representation, it is worth discussing the plant model of Fig. 2 in more detail. This plant model may take two different forms. The first and most general form is simply a set of equations which map previous state and current control effort to next state: $\mathbf{y}_k = f_p(\mathbf{u}_k, \mathbf{y}_{k-1})$ [40]–[42]. If this set of equations is nonlinear, training the neural network of Fig. 2 using a steepest descent algorithm results in a nonlinear state feedback controller. Thus, the steepest descent algorithms presented in this paper can be used as a general design techniques for developing nonlinear controllers.

The second form of a plant model is a feedforward neural network [12], [33], [43]. In this case, the plant model takes

[3] When the system of Fig. 2 is used for real-world control, the plant model is replaced by the actual plant.

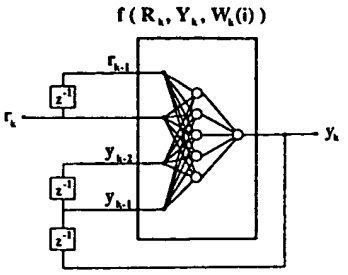


Fig. 3. neural network filter in the standard representation.

the form $\mathbf{y}_k = f_p(\mathbf{u}_k, \mathbf{y}_{k-1}, \mathbf{W}^p)$, where $\mathbf{W}^p$ is the weight vector of the neural network. This weight vector is derived by training the neural network on plant input-output data using the backpropagation algorithm [1], [44], [45]. Once trained, this model can be updated on-line using additional plant input-output data. Because the steepest descent algorithms presented in Section V use the plant model to update the controller, a neural network controller trained in this manner can adapt to slow changes in a plant. Therefore, the steepest descent algorithms can be used for nonlinear adaptive control.

Figure 3 shows an example of a single-input, single-output, nonlinear IIR filter in the standard representation. In this example, the external filter input, $\mathbf{r}_k$, along with a delayed version of this signal, $\mathbf{r}_{k-1}$, form one component of the input to a neural network. Delayed versions of the output, $\mathbf{y}_{k-1}$ and $\mathbf{y}_{k-2}$, form the other component of the input. The neural network in this example is assumed to have a sigmoidal hidden layer and a linear output unit.

III. Ordered Partial Derivatives

To update the weights using steepest descent, a partial derivative of the associated dynamical system must be calculated. Because a dynamical system contains feedback, the calculation of this derivative can be quite complex. The ordered partial derivative, which is a partial derivative whose constant and varying terms are defined using an ordered set of equations, provides a mathematical tool for easily finding derivatives of complex dynamical systems [6].

To define the ordered derivative, the concept of an ordered set of equations must first be introduced. Let $\{z_1, \ldots, z_i, \ldots, z_j, \ldots, z_n\}$ be a set of n variables whose values are determined by a set of n equations. This set of equations is defined to be an ordered set of equations if each variable z_i is a function only of the variables $\{z_1, \ldots, z_{i-1}\}$. Thus, the equation for any variable of an ordered set of equations can be written as

$$z_i = f_i(z_1, \ldots, z_{i-1}).$$

Because of the ordered nature of this set of equations, the variables $\{z_1, \ldots, z_{i-1}\}$ must be calculated before z_i can be computed. As an example, the following three equations form an ordered set of equations:

$$z_1 = 1 \tag{2}$$

$$z_2 = 3z_1 \tag{3}$$

$$z_3 = z_1 + 2z_2. \tag{4}$$

When calculating a partial derivative it is necessary to specify which variables are held constant and which are allowed to vary. Typically, if this is not specified, it is assumed that all variables are held constant except those terms appearing in the denominator of the partial derivative. This is the convention adopted in this paper; thus, the partial derivative of z_3 with respect to z_1, $\partial z_3/\partial z_1$, is 1.

An ordered partial derivative is a partial derivative whose constant and varying terms are determined using an ordered set of equations. The constant terms of the ordered partial derivative of z_j with respect to z_i, which is denoted $\partial^+ z_j/\partial z_i$ in order to distinguish it from an ordinary partial derivative, are $\{z_1, \ldots, z_{i-1}\}$. The varying terms are $\{z_i, \ldots, z_j, \ldots, z_n\}$. Using mathematical notation, the ordered partial derivative is defined as

$$\frac{\partial^+ z_j}{\partial z_i} = \left.\frac{\partial z_j}{\partial z_i}\right|_{\{z_1, \ldots, z_{i-1}\}\text{held constant.}}$$

Using this definition, the following two properties of the ordered derivative can easily be shown

$$\frac{\partial^+ z_{i+1}}{\partial z_i} = \frac{\partial z_{i+1}}{\partial z_i}$$

and

$$\frac{\partial^+ z_j}{\partial z_i} = 0 \text{ if } j < i.$$

When $j > i + 1$, the ordered derivative is found using either of the following two chain rule expansions:

$$\frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_i} = \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_i} + \sum_{k=i+1}^{j-1} \frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_k}\frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_i}, \tag{5}$$

and

$$\frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_i} = \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_i} + \sum_{k=i+1}^{j-1} \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_k}\frac{\partial^+ \mathbf{z}_k}{\partial \mathbf{z}_i}. \tag{6}$$

As an example, using either the first or second chain rule expansions and the ordered set of equations given in (2)–(4), the ordered partial derivative of z_3 with respect to z_1, $\partial^+ z_3/\partial z_1$, is 7.

Finally, one comment on mathematical notation, throughout this paper it is assumed that a partial derivative of the form $\partial \mathbf{a}/\partial \mathbf{b}$, where $\mathbf{a} \in R^{[A\times 1]}$ and $\mathbf{b} \in R^{[B\times 1]}$, is a matrix of the form $R^{[A\times B]}$.

IV. Epochwise and On-Line Training

Steepest descent algorithms adapt the weights of a discrete-time dynamical system by minimizing an error function. The definition of this error function is dependent upon whether the system is being trained in an epochwise or on-line mode. In this section, both epochwise and on-line training algorithms are defined. The epochwise and on-line error functions as well as their associated weight update equations are also presented.

A. Epochwise Training Algorithms

An *epoch* is an iteration to iteration cycling of a discrete-time dynamical system from initial to final iteration ($k = k_f$). An *epochwise training algorithm* is any algorithm in which training takes place after each epoch or after a series of epochs.

Epochwise systems are encountered much more frequently in control applications than in filtering applications. (Epochwise controllers are often referred to as terminal controllers.) In control applications, the error function used to train epochwise controllers is determined by the type of control desired. If the controller is required only to drive the plant to a desired final state $\mathbf{d}_{k_f}$, then the error function

$$E = \frac{1}{2}(\mathbf{d}_{k_f} - \mathbf{y}_{k_f})^T(\mathbf{d}_{k_f} - \mathbf{y}_{k_f}) \tag{7}$$

may be used. However, if the plant is to follow a trajectory given by a set of desired states, $\{\mathbf{d}_0, \ldots, \mathbf{d}_{k_f}\}$, then the error function

$$E = \sum_{k=0}^{k_f} \frac{1}{2}(\mathbf{d}_k - \mathbf{y}_k)^T(\mathbf{d}_k - \mathbf{y}_k) \tag{8}$$

can be used. The error function for filtering applications is also determined by the required performance. Most often in filtering applications the output is to track a desired response at each iteration. In such cases, the error function of (8) may be used to adapt the weights.

Utilizing steepest descent, epochwise algorithms update the weights using

$$w(i+1) = w(i) - \mu \frac{\partial^+ E}{\partial w(i)} \tag{9}$$

where μ, the learning rate, is a suitably chosen positive constant. Notice that the ordered partial derivative of the error with respect to the weights is used to update the weights. The update rule generates a new weight vector $\mathbf{W}(i+1)$ from the vector $\mathbf{W}(i)$. If either of the error functions defined in (7) or (8) are used, the weights are updated after each epoch. Therefore, if weight vector $\mathbf{W}(0)$ is used for the first epoch, weight vector $\mathbf{W}(i-1)$ is used for the ith epoch.

B. On-Line Training Algorithms

If the weight update of an algorithm at the current iteration k' depends only on the states of the system at iterations $\{0, \ldots, k'-1, k'\}$, then the algorithm is defined to be an on-line training algorithm. The implied dependence only upon the current and past values of the system often allows the weight updates to be computed in real-time. The key difference between on-line and epochwise training algorithms is that an on-line algorithm adapts the weights of the system as it runs while an epochwise training algorithm only updates the weights after the final iteration.

On-line training algorithms can be used in both adaptive control and filtering applications. In control applications, on-line training allows a controller to either track slow changes in plant dynamics or adapt to unknown plant characteristics. In adaptive filtering applications, on-line training can be used to develop nonlinear filters in real-time as the system runs. This can usually be accomplished whenever a desire response in some form is accessible [38].

In both on-line controller and filtering applications, an error is usually defined at each iteration. At the forward iteration k', the error $E_{k'}$ is usually a function of the desired response vector $\mathbf{d}_{k'}$ and the output vector $\mathbf{y}_{k'}$. For on-line applications, it is common to use the error function

$$E_{k'} = \frac{1}{2}(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T(\mathbf{d}_{k'} - \mathbf{y}_{k'}). \tag{10}$$

Using this error function, it is possible and usually preferable to update the weights at each iteration. Because of the update at each iteration, calculation of an exact error gradient is not possible. Instead, an approximation of the error gradient must be used to update the weights. Therefore, the on-line update rule at iteration k' is expressed as

$$w(k'+1) = w(k') - \mu \frac{\partial^+ \widehat{E}_{k'}}{\partial w(k')}, \tag{11}$$

where μ is a suitably chosen positive constant and $\partial^+ \widehat{E}_{k'}/\partial w(k')$ is the approximate error gradient.

V. Algorithms

In this section the general algorithms for adapting discrete-time dynamical systems, both backsweep and recursive, are described. Because dynamical systems may operate in either epochwise or on-line modes, two versions of each of these algorithms are presented. In addition, specific algorithms for adapting discrete-time dynamical systems composed solely of neural networks, which we shall refer to as dynamical neural networks, are also described. These algorithms may be used directly to train neural network controllers and filters. Finally, in order to select the appropriate algorithm for training a dynamical neural network, it is necessary to understand the computational and storage requirements of the various methods. Therefore, an analysis of these requirements for each of the dynamical neural network algorithms is included.

The differences between the backsweep and recursive algorithms can be traced to the differences in the derivations of the epochwise algorithms. (The on-line versions of both algorithms are easily derived from the epochwise versions. The detailed derivations of the epochwise algorithms are included in Appendices A and B.) As shown in Appendix A, computing the error derivative, $\partial^+ E/\partial w(i)$, using repeated application of the first chain rule expansion, (5), results in the epochwise backsweep algorithm. Calculating $\partial^+ E/\partial w(i)$ using the second chain rule expansion, (6), leads to the epochwise recursive algorithm. Even though both types of algorithms perform steepest descent, the characteristics of the two classes of algorithms are quite different.

As is shown later in this section, the backpropagation algorithm is used in combination with either the backsweep or recursive algorithms when training dynamical neural networks. Therefore, an understanding of the computational requirements of the backpropagation algorithm is necessary in order to derive the complexity of the dynamical neural network algorithms. The backpropagation algorithm consists of a forward

propagation through a neural network, a backward propagation, and a weight update for each training epoch [1]. Each of these computations requires on the order of Q multiplications and additions, where Q is the number of weights in the network. Therefore, the computational complexity of a weight update using the backpropagation algorithm is on the order of $3Q$ operations. (The term *operation* shall refer to one multiplication and addition.)

A. The Epochwise Backsweep Algorithm

The epochwise backsweep algorithm is derived in full detail in Appendix A. The algorithm, its implementation, and its computational and storage requirements are presented in this section. The algorithm is implemented by performing the following at each epoch:

1) *Create an Epoch.* Given the initial conditions $\mathbf{R}_0$ and $\mathbf{Y}_0$, the weight vector $\mathbf{W}(i)$ and the set of external inputs $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_{k_f}\}$, cycle the two equations

$$\begin{aligned} \mathbf{W}_k(i) &= \mathbf{W}(i), \\ \mathbf{y}_k &= f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \end{aligned} \tag{12}$$

starting from the initial iteration ($k = 0$) to the final iteration ($k = k_f$). This process forms an epoch of the discrete-time dynamical system.

2) *Calculate the Lagrange mulitipliers.* The Lagrange multipliers, which are defined as

$$\lambda_k = \frac{\partial^+ E}{\partial \mathbf{y}_k},$$

are used to simplify the calculation of the error gradient. Given an epochwise error function, the Lagrange multipliers are computed by cycling

$$\lambda_k = \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=1}^{L} \lambda_{k+j} \frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k} \quad k = k_f - 1, \ldots, 0 \tag{13}$$

backwards from iteration $k = k_f - 1$ to $k = 0$. The following two equations are used to establish the initial conditions for (13):

$$\lambda_{k_f} = \frac{\partial E}{\partial \mathbf{y}_{k_f}},$$

$$\lambda_{k_f+1}, \ldots, \lambda_{k_f+L} = 0.$$

3) *Compute the Error Derivative.* Given the Lagrange multipliers, the ordered partial derivative of the error with respect to a weight is calculated using

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(i)}. \tag{14}$$

The term $\partial \mathbf{y}_k / \partial w_k(i)$ is the derivative of the output at iteration k with respect to one of the weights used in the dynamical system at iteration k.

4) *Update the weights.* The weights are updated using the epochwise steepest descent rule (9).

The epochwise backsweep algorithm can be used to train any discrete-time dynamical system provided that the derivatives of the output $\mathbf{y}_k$ with respect to the weights $\mathbf{W}(i)$ and recurrent inputs $\mathbf{Y}_k$ exist. For any general system, the term $\partial E / \partial \mathbf{y}_k$ of (13) is found by differentiating the error function. The terms $\partial \mathbf{y}_{k+j} / \partial \mathbf{y}_k$ of (13) and $\partial \mathbf{y}_k / \partial w_k(i)$ of (14) can both be calculated by differentiating the output, which is given by (12).

1) Backpropagation Through Time: If the function $f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}(i))$ is implemented by a feedforward neural network, as is the case for the IIR neural filter of Fig. 3 or the neural controller of Fig. 2 with a plant model represented by a neural network, it is possible to use a computationally efficient method to calculate the Lagrange multipliers of (13) and the error gradients of (14). This method, which is referred to as the backpropagation through time algorithm [1], [12]–[18], is an example of the lowest layer of the theory on training adaptive dynamical systems.

The efficiency of this method is achieved by using the backpropagation algorithm to compute partial deviatives which are of the form

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w} = \lambda \frac{\partial \mathbf{y}}{\partial w}. \tag{15}$$

Whenever the vector $\mathbf{y}$ of (15) represents the output of a feedforward neural network, the error derivative can be efficiently computed by "backpropagating" the vector λ through the feedforward neural network [1]. In the dynamical neural network case, since $\mathbf{y}_k$ is the output of a feedforward network with weights $\mathbf{W}_k(i)$ and inputs $\mathbf{R}_k$ and $\mathbf{Y}_k$, the backpropagation algorithm can be used in combination with the backsweep technique to efficiently compute the weight updates. By backpropagating the vector λ_k through the feedforward network of iteration k, the product $\lambda_k \partial \mathbf{y}_k / \partial w_k(i)$ of (14) can be efficiently computed. In addition, by backpropagating the vector λ_k to the recurrent inputs of this netwqrk, the set

$$\{\lambda_k \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-1}}, \ldots, \lambda_k \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-L}}\},$$

which is used in (13), can be efficiently computed.

To minimize the number of backpropagations required to calculate the epochwise error derivative, in the calculation of the vector λ_k in Step 2, the vector λ_{k+1} should be backpropagated all the way through the neural network of iteration $k + 1$, and the results should be stored in memory. Using this approach only $k_f + 1$ backpropagations are required.

If the function $f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}(i))$ is implemented only partially by a neural network, as is the case if the plant model of Fig. 2 is implemented by a set of equations, the backpropagation algorithm can still be used to reduce the computational requirements of calculating the error derivative. In such systems, vector-derivative products associated with the feedforward neural network portion must be computed at each iteration as part of the calculation of the terms of the sums of (13) and (14). By using the backpropagation algorithm for these calculations, the computational requirements can be minimized. Thus, a combination of the backsweep and backpropagation algorithms can also be used to train a neural

network controller when the plant model is implemented by a set of equations.

2) Computational and Storage Requirements of Backpropagation Through Time: In this section the computational and storage requirements of training dynamical neural networks using the backpropagation through time algorithm are discussed. This algorithm is based upon $k_f + 1$ forward and backward propagations through the dynamical neural network (Steps 1 and 2) for each epoch. Because each of these propagations requires on the order of Q multiplications and additions, where Q is the number of weights, $2(k_f+1)Q$ operations are needed to calculate the k_f partial error derivatives of the right hand side of (14). Summing these derivatives and using the result to update the weights (9) requires approximately $(k_f + 1)Q$ additional operations. The total number of multiplications and additions for each weight update using the backpropagation through time algorithm is

$$C_{EB} \approx 3(k_f + 1)Q.$$

The storage requirement of the algorithm is composed of two different components. First, the weights and their associated error derivatives need to be stored in memory for efficient computation. These terms require $2Q$ memory locations. Secondly, the output vector $\mathbf{y}_k$ and external input vector $\mathbf{r}_k$ must be stored at each iteration of the forward propagation for use in the calculation of the error derivatives. Storing only the input and output vectors of each iteration minimizes the storage requirement while increasing the computation requirements by at most $(k_f + 1)Q$. This increase occurs because the internal nodes of the neural network must be recalculated in the backward sweep if these values are not stored. The external input $\mathbf{r}_k$ is a vector of length M while the output $\mathbf{y}_k$ is a vector of length N. Thus, $(k_f+1)(M+N)$ memory locations are required for the external inputs and outputs. Adding the two components together, the minimal storage requirement of the algorithm is

$$S_{EB} \approx (k_f + 1)(M + N) + 2Q.$$

B. The On-Line Backsweep Algorithm

The on-line version of the backsweep algorithm is based upon the concept of introducing artificial epochs into the system and then using the epochwise algorithm to update the weights. An artificial epoch is created by assuming that the current iteration k' is the final iteration of an artificial epoch. The first iteration of the artificial epoch is assumed to be $k'-T$, where T is a positive integer constant. If $k' - T < 0$, then the first iteration is $k = 0$.

Calculating the error derivative using an artificial epoch of length $T + 1$ leads to an approximation of the true derivative. Thus, in the on-line case the weights must be updated using the following approximate error gradient

$$\frac{\partial^+ \widehat{E}_{k'}}{\partial w(k')} = \sum_{k=k'-T}^{k'} \hat{\lambda}_k \frac{\partial \hat{\mathbf{y}}_k}{\partial w_k(i)},$$

where $\hat{\mathbf{y}}_k$ and $\hat{\lambda}_k$ are the approximate outputs and Lagrange multipliers respectively. (These are defined below.) It should be noted that because of the weight updates at each iteration, the terms $\partial \hat{\mathbf{y}}_k / \partial w_k(i)$ generally becomes less accurate as k is decreased from k'. To reduce these greater inaccuracies, it is possible to use exponential weighting (See Step 3 below) [46]. The on-line backsweep algorithm is implemented by performing the following at each iteration:

1) *Iterate the dynamical system.* At the current iteration k', given the input vector $\mathbf{R}_{k'}$, feedback output vector $\mathbf{Y}_{k'}$ and weight vector $\mathbf{W}_{k'}(k')$, iterate the dynamical system once using

$$\mathbf{y}_{k'} = f(\mathbf{R}_{k'}, \mathbf{Y}_{k'}, \mathbf{W}_{k'}(k')).$$

The weight vector $\mathbf{W}_{k'}(k')$ is an updated version of the weights used at the previous iteration.

2) *Calculate the approximate Lagrange multipliers.* The Lagrange multipliers cannot be calculated precisely because of the weight changes at each iteration. An approximation to the Lagrange multipliers can be found by backward iterating the following equation:

$$\hat{\lambda}_k = \frac{\partial E_{k'}}{\partial \hat{\mathbf{y}}_k} + \sum_{j=1}^{L} \hat{\lambda}_{k+j} \frac{\partial \hat{\mathbf{y}}_{k+j}}{\partial \hat{\mathbf{y}}_k} \quad k = k' - 1, \ldots, k' - T, \tag{16}$$

where

$$\lambda_k \approx \hat{\lambda}_k = \frac{\partial^+ E_{k'}}{\partial \hat{\mathbf{y}}_k}.$$

The approximate output $\hat{\mathbf{y}}_k$ is defined as

$$\mathbf{y}_k \approx \hat{\mathbf{y}}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(k')). \tag{17}$$

In (17), $\mathbf{y}_k$, which is calculated using the weight vector $\mathbf{W}_k(k)$, is approximated by $\hat{\mathbf{y}}_k$, which is computed using $\mathbf{W}_k(k')$. Obviously, the smaller the difference between the weights at iteration k and those at iteration k' the more accurate this approximation will be and thus the more accurate the calculation of the Lagrange multipliers will be in (16). In the on-line case, it is common to use the mean squared error as the error function (10). In such cases, the boundary conditions of (16) take the form

$$\hat{\lambda}_{k'} = -(\mathbf{d}_{k'} - \mathbf{y}_{k'})$$

$$\hat{\lambda}_{k'+1}, \ldots, \hat{\lambda}_{k'+L} = 0$$

$$\frac{\partial E_{k'}}{\partial \hat{\mathbf{y}}_{k'-T}}, \frac{\partial E_{k'}}{\partial \hat{\mathbf{y}}_{k'-T+1}}, \ldots, \frac{\partial E_{k'}}{\partial \hat{\mathbf{y}}_{k'-1}} = 0.$$

3) *Calculate the approximate error derivative.* The approximate error derivative, with exponential weighting, is calculated using

$$\frac{\partial^+ \widehat{E}_{k'}}{\partial w(k')} = \sum_{k=k'-T}^{k'} \alpha^{k'-k} \hat{\lambda}_k \frac{\partial \hat{\mathbf{y}}_k}{\partial w_k(i)}, \tag{18}$$

where the constant $0.0 < \alpha < 1.0$ is the exponential weighting coefficient. The weighting causes the less precise terms of (18) to contribute less to the overall error derivative.

4) *Update the weights.* The weights are adapted at each iteration using the on-line update rule of (11).

One difficulty associated with the on-line backsweep algorithm is the selection of appropriate values for the constants T and α. Like the selection of the learning rate, μ, there are no analytic procedures for choosing these constants. Instead, the selection should be based upon knowledge of the dynamical system, desired convergence rate and required misadjustment upon convergence.

1) On-Line Backpropagation Through Time: Once again, the on-line algorithm can be used to train any discrete-time dynamical system whose output $\mathbf{y}_{k'}$ is differentiable with respect to the weights $\mathbf{W}(k')$ and recurrent inputs $\mathbf{Y}_{k'}$. Thus, the algorithm can be used for on-line adaptation of neural network IIR filters and controllers. If the function $f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}(i))$ is implemented by a feedforward neural network, a combination of the backpropagation and on-line backsweep algorithms, which is referred to as on-line (or truncated) backpropagation through time [27], [47], can be used to update the weights. Recall that in the epochwise version, when a dynamical neural network is trained, it is necessary to backpropagate the vector λ_{k+1} through the feedforward neural network with output $\mathbf{y}_{k+1}$ in order to calculate λ_k in Step 2. In the on-line version, the vector $\hat{\lambda}_{k+1}$ must also be backpropagated through the neural network with output $\hat{\mathbf{y}}_{k+1}$ in order to compute $\hat{\lambda}_k$. However, before this backpropagation can occur, a forward propagation of $\mathbf{R}_{k+1}$ and $\mathbf{Y}_{k+1}$ through the feedforward neural network with weight vector $\mathbf{W}(k')$ must be performed in order to establish the appropriate conditions for the backpropagation. Therefore, at each iteration of Step 2 of the on-line backpropagation through time algorithm, $\hat{\mathbf{y}}_{k+1}$ is first computed by a forward propagation, $\hat{\lambda}_{k+1}$ is then backpropagated and finally $\hat{\lambda}_k$ is calculated using (16).

2) Computational and Storage Requirements: The computational complexity and storage requirements of the on-line backpropagation through time algorithm for a dynamical neural network with Q weights are easily derived from those of the epochwise algorithm. Because an artificial epoch of length $T+1$ is used to update the weights in the on-line version, the computational and storage requirements can be found by simply making the substitutions $T = k_f$ into the epochwise requirements. Thus, the computational complexity at each iteration of the on-line backsweep algorithm is

$$C_{OB} \approx 3(T+1)Q,$$

while the storage requirement is

$$S_{OB} \approx (T+1)(M+N) + 2Q.$$

C. Epochwise Recursive Algorithm

The epochwise recursive algorithm is derived in full detail in Appendix B. As shown in this derivation, the algorithm is developed using the second chain rule expansion, (6). The algorithm is implemented by performing the following at each epoch:

1) *Calculations performed at each iteration of the epoch.* The epochwise recursive algorithm is based upon recursively updating derivatives of the output and error. These updates are computed using the following sequence of calculations at each iteration of the epoch.

   a) *Select the weight vector and iterate the dynamical system.* At each iteration k, the weight vector is selected as

   $$\mathbf{W}_k(i) = \mathbf{W}(i),$$

   and the dynamical system is iterated once using

   $$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)).$$

   The dynamical system is initialized by selecting values for $\mathbf{R}_0$ and $\mathbf{Y}_0$.

   b) *Recursively calculate the output derivative.* At each iteration k, the output derivative is calculated using

   $$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \sum_{j=1}^{L} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-j}} \frac{\partial^+ \mathbf{y}_{k-j}}{\partial w(i)}. \tag{19}$$

   If $k < L$, where L is the maximum delay in the discrete-time dynamical system, the initial conditions

   $$\frac{\partial^+ \mathbf{y}_{-1}}{\partial w(i)}, \cdots, \frac{\partial^+ \mathbf{y}_{-L}}{\partial w(i)} = \mathbf{0}$$

   are used in (19). For $k - j \geq 0$, the values of the terms $\partial^+ \mathbf{y}_{k-j}/\partial w(i)$ should be stored in memory having been calculated using (19) at previous iterations.

   c) *Recursively calculate the error derivative.* As shown in Appendix B, the epochwise error derivative is

   $$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)},$$

   which may be recursively calculated using

   $$S_k = S_{k-1} + \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)}. \tag{20}$$

   This equation is initialized by $S_{-1} = \mathbf{0}$. It is easy to show that the epochwise error derivative is

   $$\frac{\partial^+ E}{\partial w(i)} = S_{k_f}.$$

   Therefore, to calculate the error gradient at each iteration k, S_k should be updated using (20). Of course, the term $\partial^+ \mathbf{y}_k/\partial w(i)$ of the recursive calculation (20) is obtained from Step 1-b).

2) *Update the weights.* After the final iteration of the epoch, the weights are adapted using the error derivative S_{k_f} and the epochwise update rule of (9).

1) Epochwise Recursive Backpropagation: The epochwise recursive algorithm can be used to train any discrete-time dynamical system provided that the derivatives of the output with respect to both the weights and recurrent inputs exist. Except for the calculation of the output derivative in Step 1-b), implementation of the algorithm is fairly simple; therefore, we shall only discuss calculation of the output derivative.

To update the output derivative at each iteration using (19), it is necessary to compute the direct output derivative $\partial \mathbf{y}_k / \partial w_k(i)$ and the Jacobian matrix $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$. Different techniques, depending upon the form of the dynamical system, can be used to calculate these terms. As shown below, if the structure of the discrete-time dynamical system is a neural network, these two components can be found using N backpropagations of N appropriately selected vectors $\{\lambda_1, \ldots, \lambda_n, \ldots, \lambda_N\}$. ($N$ is the number of outputs of the system in the standard representation.) The N vectors take the form

$$\lambda_{nj} = \begin{cases} 1 & \text{if } n = j \\ 0 & \text{otherwise} \end{cases}$$

where λ_{nj} is the jth element of the row vector $\lambda_n \in R^{[1 \times N]}$. Each vector has only one nonzero component, which appears in the $n = j$ column. Because $\mathbf{y}_k$ is the output vector of a feedforward neural network, the backpropagation of the vector λ_n through the neural network at iteration k results in the calculation of the output derivative of the nth output, as indicated by

$$\lambda_n \frac{\partial \mathbf{y}_k}{\partial w_k(i)} = \frac{\partial y_k(n)}{\partial w_k(i)}.$$

Furthermore, as shown by

$$\lambda_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{Y}_k} = \frac{\partial y_k(n)}{\partial \mathbf{Y}_k},$$

backpropagating the vector λ_n to the input nodes results in the calculation of the nth row of the Jacobian matrix. We can conclude that the direct output gradient $\partial \mathbf{y}_k / \partial w_k(i)$ and Jacobian matrix $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$ can be computed by backpropagating $\{\lambda_1, \ldots, \lambda_n, \ldots, \lambda_N\}$ through the kth iteration of the dynamical network. Using the epochwise recursive algorithm with the direct output gradient and Jacobian matrix calculated in this manner is referred to as epochwise recursive backpropagation.

2) Complexity and Storage Requirements: The epochwise recursive backpropagation algorithm is based upon the following three operations being performed at each iteration: a forward propagation of the dynamical system, a recursive update of the output derivative and a recursive update of the error gradient. When a dynamical neural network with Q weights is trained, the computational complexity of the $k_f + 1$ forward propagations is $(k_f + 1)Q$. The computational complexity of calculating the output derivative is determined by the number of backpropagations required for an epoch and the complexity of the recursive output derivative update. The two terms $\partial \mathbf{y}_k / \partial w_k(i)$ and $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$ must be computed using N backpropagations at each of the $k_f + 1$ iterations of the epoch. Therefore, the computational complexity of calculating these terms for all iterations of the epoch is $(k_f + 1)NQ$. Given the results of the backpropagations, the output derivatives for each weight are calculated as shown in (19). This calculation requires L matrix-vector multiplications and L vector-vector additions per iteration to compute the output derivatives for each weight. Therefore, the computational requirements of calculating all the output derivatives for an epoch is $(k_f + 1)(N^2LQ + NQ)$. The error derivative for each weight is computed using the recursive equation of Step 1-c). This calculation requires N multiplications and additions per iteration. Therefore, the computational requirements of implementing Step 1-c) at each iteration for all weights is $(k_f + 1)NQ$. The total computational requirements of the epochwise recursive backpropagation algorithm for a dynamical neural network is

$$C_{ER} \approx (k_f + 1)(N^2L + 2N + 1)Q.$$

The storage requirements are determined primarily by the need to store the $L + 1$ output derivative vectors for each weight (19). The total storage requirements of these output derivatives is $(L+1)NQ$. In addition to the output derivatives, $2Q$ memory locations are need to store the weights and error derivatives. Hence, the storage requirement of the epochwise recursive algorithm is

$$S_{ER} \approx (L + 1)NQ + 2Q.$$

D. On-Line Recursive Algorithm

The calculation of the output gradient at each iteration in the epochwise version of the recursive algorithm depends only upon the current and past values of the dynamical system. The lack of dependence on future values of the system in calculating the output gradient makes the algorithm attractive for on-line implementation. The on-line recursive algorithm is implemented by performing the following at each iteration:

1) *Select the weight vector and iterate the dynamical system.* At the current iteration k', the weight vector is selected as

$$\mathbf{W}_{k'}(k') = \mathbf{W}(k'),$$

and the dynamical system is iterated once using

$$\mathbf{y}_{k'} = f(\mathbf{R}_{k'}, \mathbf{Y}_{k'}, \mathbf{W}_{k'}(k')).$$

The dynamical system is initialized by selecting values for $\mathbf{R}_0$ and $\mathbf{Y}_0$.

2) *Recursively calculate an approximate output derivative.* The approximate output derivative is computed at each iteration using

$$\frac{\partial^+ \widehat{\mathbf{y}_{k'}}}{\partial w(k')} = \frac{\partial \mathbf{y}_{k'}}{\partial w_{k'}(k')} + \sum_{j=1}^{L} \alpha^j \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{y}_{k'-j}} \frac{\partial^+ \widehat{\mathbf{y}_{k'-j}}}{\partial w(k'-j)} \quad (21)$$

where $0.0 < \alpha < 1.0$. The precise output derivative cannot be calculated because of the weight changes at each iteration. The exponential weighting term α is-

TABLE I
COMPUTATION AND STORAGE REQUIREMENTS.

ZZ	Epochwise algorithms		Online algorithms	
Requirements	Backpropagation through time	Recursive backpropagation	Backpropagation through time	Recursive backpropagation
Computational	$3(k_f+1)Q$	$(k_f+1)(N^2L+3N+1)Q+Q$	$(2T+3)Q$	$(N^2L+N(L+3)+2)Q$
Storage	$2Q+(k_f+1)(M+N)$	$(L+1)NQ$	$2Q+(T+1)(M+N)$	(L+1)NQ

introduced to reduce the effects of the less accurate terms of the summation [46]. Equation (21) is initialized using

$$\frac{\partial^+\widehat{\mathbf{y}_{-1}}}{\partial w(-1)}, \cdots, \frac{\partial^+\widehat{\mathbf{y}_{-L}}}{\partial w(-L)} = \mathbf{0}.$$

The second term in the summation of (21) is either stored in memory having been calculated using this same equation at a previous iteration or determined by the initial conditions.

3) *Recursively calculate an approximate error derivative.* Assuming the mean squared error is used as the error criterion, the approximate error derivative can be calculated using

$$\frac{\partial^+\widehat{E}}{\partial w_{k'}(k')} = -(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T \frac{\partial^+\widehat{\mathbf{y}_{k'}}}{\partial w_{k'}(k')}. \tag{22}$$

Once again, the error derivative cannot be precisely calculated because of the weight changes at each iteration.

4) *Update the weights.* At the end of each iteration, the weights are adapted using the on-line update rule of (11).

1) On-Line Recursive Backpropagation: The on-line recursive algorithm can be used to adapt the weights of neural network IIR filters and controllers. As discussed in the previous subsection, in these cases the two terms $\partial \mathbf{y}_{k'}/\partial w_{k'}(k')$ and $\partial \mathbf{y}_{k'}/\partial \mathbf{Y}_{k'}$ can be calculated using N backpropagations at each iteration. This technique of combining the on-line recursive technique and the backpropagation algorithm is referred to as the on-line recursive backpropagation algorithm [15], [16], [29]–[32].

The computational requirement of the on-line recursive backpropagation algorithm follows almost immediately from those of the epochwise version. The only differences between the two versions of the recursive algorithm are the exponential weighting of the output derivative calculation (21) and the need for weight updates at each iteration in the on-line case. The exponential weighting adds an additional LNQ computations to the calculation of the output derivatives, therefore, using the results of the previous subsection, the output derivative computation requires $(N^2L+N(L+1))Q$ operations. At each iteration, the approximate error gradient calculation (22) requires NQ multiplications and additions while the weight update requires Q operations. We conclude that the computational complexity per iteration of the on-line recursive algorithm is

$$C_{OR} \approx (N^2L+N(L+2)+2)Q.$$

The exponential weighting and weight update of the on-line algorithm does not introduce any additional storage requirements. Therefore, the storage requirement is identical to that of the epochwise version:

$$S_{OR} \approx (L+1)NQ + 2Q.$$

VI. COMPARISON OF ALGORITHMS

The computational and storage requirements of the dynamical neural networks training algorithms presented in Section V are outlined in Table I. In this section, we use the results summarized in this table to compare the algorithms. It should be noted that although we are concerned only with the algorithms for dynamical neural networks in this section, many of the conclusions arrived at in this section apply equally well to the associated generalized algorithms.

Starting with the epochwise algorithms, we find the backpropagation through time technique to be computationally superior to the recursive method. The inefficiency of the recursive method is a result of the matrix-vector multiplication in the output derivative calculation of (19). Although the recursive algorithm is inefficient, it has the advantage of fixing the storage requirements to $(L+1)NQ+2Q$, which is independent of the number of iterations in an epoch. In some cases, where the total number of iterations k_f+1 in an epoch is large compared to the number of weights Q, the epochwise recursive algorithm may be advantageous to use for this reason. However, in most cases the total number of iterations is small compared to the number of weights, and the storage requirements favor the use of the epochwise backsweep algorithm. Because the backsweep algorithm is computationally more efficient and requires less storage in most cases, it should be used to train epochwise dynamical neural networks, including neural network controllers and filters.

The ratio of the computational requirements of the two on-line algorithms,

$$\frac{C_{OB}}{C_{OR}} \approx \frac{3(T+1)Q}{(N^2L+N(L+2)+2)Q}, \tag{23}$$

can be used to select the most efficient on-line method. Unlike in the epochwise case, neither algorithm is universally computational superior to the other. Instead, the selection of the most efficient on-line method depends on the number of outputs N, the maximum delay in the feedback loop L and the window length of the backsweep algorithm T. In addition to computational efficiency, the storage requirements should also be used to select the appropriate on-line algorithm. On the basis of the storage requirements alone, the on-line backsweep algorithm is preferable to the on-line recursive technique when $(T+1)(N+M) < (L+1)NQ$. For almost all dynamical neural network systems, this inequality will hold and the storage

requirements will favor the use of the on-line backpropagation through time algorithm.

The selection of an on-line algorithm for training a neural network IIR filter should be based on the number of outputs and the maximum delay in the feedback loop of the dynamical system. IIR filters with one output and a moderate number of feedback delays are commonly used in adaptive systems. For such filters, the computational requirements favor use of the on-line recursive algorithm while the storage requirements favor use of the on-line backsweep algorithm. However, because the computational requirements are often more important than the storage requirements, the on-line recursive algorithm is recommended for adapting single output neural network filters. (It should be pointed out that on-line recursive techniques for adapting single output linear IIR filters have been well-studied [38], [39], [48] while we are unaware of any attempts to use the on-line backsweep algorithm for such filters.) While the recursive algorithm is preferable for training single output filters, the backsweep algorithm is better suited for training multidimensional filters. In the multidimensional case, the number of outputs is greater than one and the backsweep algorithm is therefore computationally superior. Summarizing the results above, the recursive algorithm is found to be superior for training single output IIR neural network filters, while the backpropagation through time algorithm is found to be better suited for adapting epochwise and multidimensional filters.

The selection of an on-line algorithm for training neural network controllers is also based on the number of outputs and the maximum delay in the feedback loop of the dynamical system. For control systems in their standard representation, the number of outputs N must be greater than or equal to the number of states in the plant. Generally, the number of states is larger than one. The maximum delay in the feedback loop in most control systems is one. Under these circumstances, the computational requirements tend to support use of the backpropagation through time algorithm. Because the storage requirements will also generally favor use of the on-line backpropagation through time algorithm, this algorithm is preferable for on-line training of neural network controllers. Given these observations along with those above for the epochwise algorithms, one can conclude that the backpropagation through time algorithms are preferable for training both epochwise and on-line controllers.

VII. Examples

In this section, two examples that illustrate the uses of the dynamical system training algorithms are presented. The first example demonstrates the use of the algorithms for nonlinear controller design. In this example, a neural network is trained using the epochwise backsweep algorithm to provide the steering angle of a boat which is placed in a river with a nonlinear current. By providing the proper steering angle, the neural network guides the boat across the river to a designated dock position. The second example illustrates the use of the on-line recursive algorithm for adaptive filtering. In this example, an adaptive noise canceller is trained to eliminate noise from a corrupted signal.

A. Nonlinear Control Example

In this example, a boat is initially placed in a river, which is 200 ft wide, within a region 100 ft upstream or downstream of a dock. The boat is powered by a constant thrust motor which is also used to point the boat in any desired direction. Starting from any initial position, it is desired to maneuver the boat to a dock, which is located on one shore of the river. Maneuvering the boat to the dock is made difficult by the stream's nonlinear current.

At iteration k, let x_k denote the distance from the center of the boat to the shore with the dock. Let y_k denote the distance of the center of the boat upstream or downstream of the dock. Assuming the current to be a function only of the distance from the shore, x_k, the equations of motion for the boat are

$$x_{k+1} = x_k + 10cos(u_k) \tag{24}$$

$$y_{k+1} = y_k + 10sin(u_k) + f_c(x_k). \tag{25}$$

where u_k, the orientation of the boat given in radians, is the control signal, and $f_c(x_k)$, the influence of the current on the boat, is given in feet per iteration. The current, which is parabolic in nature with the greatest force in the middle of the stream at $x = 100$, is given by the following equation:

$$f_c(x_k) = 7.5\left(\frac{x_k}{50} - \left(\frac{x_k}{100}\right)^2\right).$$

The control signal is supplied by the output of a three layer neural network. The first layer contains the two inputs, x_k and y_k, which are the states of the system. The hidden layer contains ten sigmoidal units which are fully connected to the inputs and a bias. The output layer, which is linear, is fully connected to the hidden layer and the bias.

The boat system operates in an epochwise manner with the initial position determined randomly during training and the final position specified as the iteration prior to the boat hitting the dock's shore. For reasons discussed in Section VI, the epochwise backsweep algorithm was selected for training the neural network controller. The plant model was implemented by (24), (25). To drive the boat to the dock at the final iteration, the following error function was used:

$$E = (x_d - x_{k_f})^2 + (y_d - y_{k_f})^2,$$

where x_d is the x position of the dock, and y_d is the y position of the dock.

To train the neural network controller, 4000 training epochs were required ($\mu = 0.0001$). After training was completed, four demonstration epochs, which are shown in Fig. 4, were run. In the lower portion of Fig. 4, the current is shown as a function of x. (To show the boat graphically, it was necessary to move the two shores outward a distance equal to half the boat length. For this reason, the current near both shores is shown as zero.) The four demonstration epochs show that by using the backpropagation through time algorithm, it is possible to design a neural network controller for guiding the boat across the river.

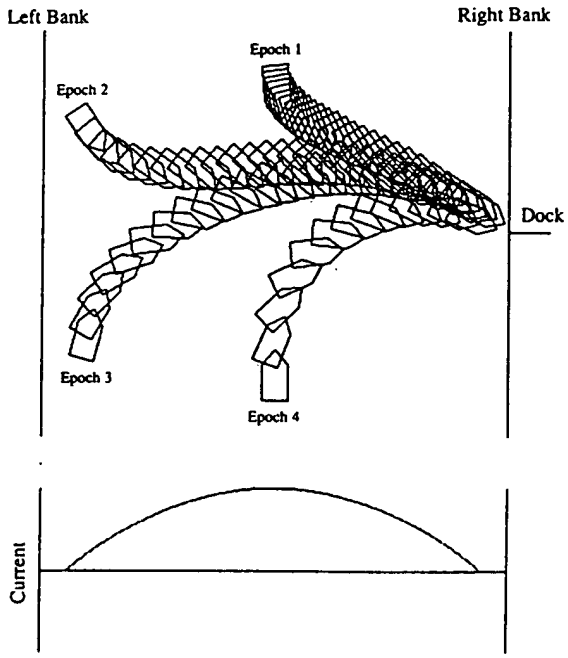


Fig. 4. Nonlinear control example.

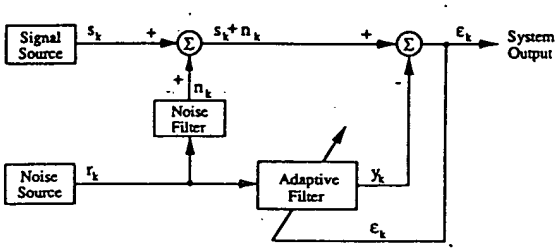


Fig. 5. Adaptive noise-cancelling system.

B. Adaptive Filtering Example

In this example, an adaptive noise cancelling system is used to reduce additive noise from a corrupted signal [38]. Fig. 5 shows an illustration of an adaptive noise cancelling system. The original signal s_k is corrupted by a noise signal n_k, which is a filtered version of the noise source r_k. The noise source, which can be sensed directly, is used as input to an adaptive filter. By training this filter using an on-line error of $E_{k'} = (s_{k'} + n_{k'} - y_{k'})^2$, the expected value of $E[(n_k - y_k)^2]$ is locally minimized [38]. Therefore, if the adaptive filter can model the noise filter, the output of the adaptive filter y_k will approximate n_k, and the noise in the system output ϵ_k will be significantly attenuated. In order to allow proper training of the adaptive filter, it is required that the noise source signal r_k and original signal s_k be uncorrelated.

If the noise filter is linear, a linear adaptive filter should be used to cancel the noise [38]. If the noise filter is nonlinear, a neural network filter can be used. In the example presented below, the noise filter was a nonlinear IIR filter, therefore, a neural network IIR filter was trained using the recursive algorithm to cancel the noise from the corrupted signal.

In the noise cancelling example, the original signal was

$$s_k = .25\cos(.4k). \tag{26}$$

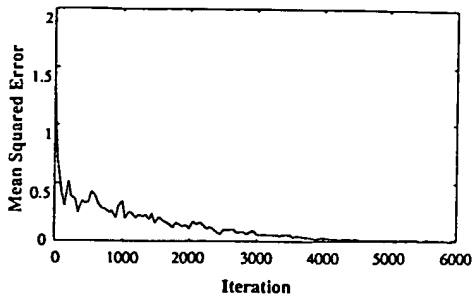


Fig. 6. Learning curve of the noise cancelling system.

The noise source signal r_k was a white uniformly distributed random variable with a range between -1.0 and 1.0. This noise source was filtered to produce n_k using the following nonlinear difference equation:

$$n_k = r_k + f_n(n_{k-1}), \tag{27}$$

where

$$f_n(n_{k-1}) = .5\exp\left(\frac{-(n_{k-1}-1.0)^2}{0.67}\right) - .5\exp\left(\frac{-(n_{k-1}+1.0)^2}{0.67}\right). \tag{28}$$

It should be noted that the noise filter contained nonlinear feedback.

The adaptive filter was implemented by a three-layer feedforward neural network. The input layer was composed of two components, the noise source signal r_k and the previous output of the adaptive filter y_{k-1}. The hidden layer was composed of 17 sigmoidal units. The first five nodes were connected to the noise source signal r_k while the remaining ten nodes were connected to the feedback signal y_{k-1}. (Each of the hidden units were connect to a bias input.) The output layer contained one linear unit which was connected to the hidden nodes and the bias input.

The filter was trained using the on-line recursive algorithm with a learn rate of $\mu = .005$ and an exponential weighting factor of $\alpha = .95$. A learning curve is shown in Fig. 6. The initial sharp decrease in the mean squared error over the first couple hundred iterations is due to relatively fast learning of the feedforward component of the filter. The slow decrease, which lasts for several thousand iterations, is due to learning the feedback component. The corrupted signal $s_k + n_k$ and the original signal s_k for iterations 5900-6000 are shown in Fig. 7. Notice that it is impossible to determine the characteristics of the original signal from the corrupted signal. The output signal ϵ_k and original signal s_k for these same iterations are shown in Fig. 8. Although the output signal is not perfect, the noise has been significantly reduced.

VIII. Conclusion

In this paper a unified framework for discrete-time dynamical system training algorithms is developed. Using this framework, it is possible to select the appropriate training algorithm for designing neural network controllers and filters.

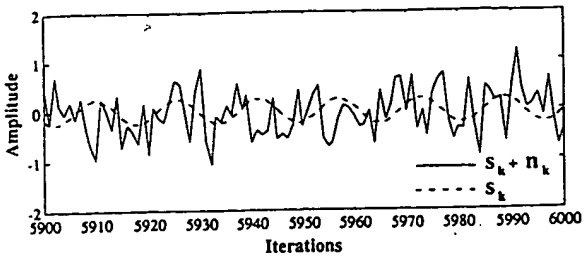


Fig. 7. Corrupted signal and original signal.

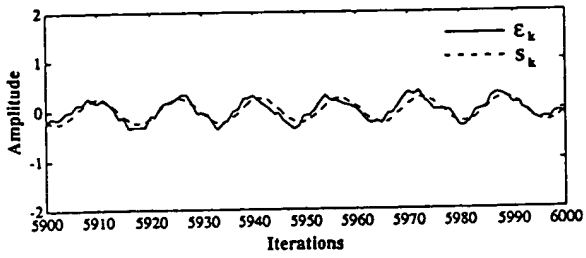


Fig. 8. Output signal and original signal.

As shown in this paper, the training of neural network controllers and filters can be accomplished using either the backpropagation through time or recursive backpropagation algorithms. The most important difference between the two techniques is the difference in computational and storage requirements. For the backpropagation through time algorithm, these requirements are independent of the number of outputs, while for the recursive algorithm, they increase quadratically with the number of outputs. Therefore, the backsweep algorithms should be used to train neural controllers because the number of outputs of a neural network control system in the standard representation is generally significantly larger than one (the number of outputs is greater than or equal to the number of states of the plant). For neural network IIR filters with a single output operating in an on-line mode, the system should be trained using the recursive algorithm, while for filters with multidimensional outputs, the system should be trained using backpropagation through time.

Appendix A: Derivation of Epochwise Backsweep Algorithm

Steps 1)-)3 of the epochwise backsweep algorithm given in Section V-A are derived in full detail below. The standard representation of Section II-A is used in this derivation. As shown below, the chain rule expansion of (5) is used repeatedly in deriving the algorithm.

Our primary goal in deriving the algorithm is to find an expression for the term $\partial^+ E/\partial w(i)$, which is used to update the weights. We begin by noting that Step 1 of the algorithm produces the following ordered set of equations:

$$\mathbf{W}_0(i) = \mathbf{W}(i) \tag{29}$$

$$\mathbf{y}_0 = f(\mathbf{R}_0, \mathbf{Y}_0, \mathbf{W}_0(i)) \tag{30}$$

$$\vdots$$

$$\mathbf{W}_k(i) = \mathbf{W}(i) \tag{31}$$

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \tag{32}$$

$$\vdots$$

$$\mathbf{W}_{k_f}(i) = \mathbf{W}(i) \tag{33}$$

$$\mathbf{y}_{k_f} = f(\mathbf{R}_{k_f}, \mathbf{Y}_{k_f}, \mathbf{W}_{k_f}(i)) \tag{34}$$

$$E = f(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{k_f}, \mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k_f}). \tag{35}$$

Using the first chain rule expansion (5) along with this set of ordered equations, the partial derivative $\partial^+ E/\partial w(i)$ may be written as

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left(\frac{\partial^+ E}{\partial \mathbf{y}_k} \frac{\partial \mathbf{y}_k}{\partial w(i)} + \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} \right). \tag{36}$$

The term $\partial E/\partial w(i)$ and the components of the vector $\partial \mathbf{y}_k/\partial w(i)$ are equal to zero because E and $\mathbf{y}_k$ are not direct functions of $w(i)$.[4] Therefore, (36) can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial w_k(i)}. \tag{37}$$

We need to find an expression for the term $\partial^+ E/\partial w_k(i)$. Once again, this expression can be found by expanding the ordered derivative using (5),

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial E}{\partial w_k(i)} + \sum_{j=k}^{k_f} \frac{\partial^+ E}{\partial \mathbf{y}_j} \frac{\partial \mathbf{y}_j}{\partial w_k(i)} + \sum_{j=k+1}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_j(i)} \frac{\partial \mathbf{W}_j(i)}{\partial w_k(i)}.$$

The term $\partial E/\partial w_k(i)$ and the components of the vector $\partial \mathbf{W}_j(i)/\partial w_k(i)$ are equal to zero. The term $\partial \mathbf{y}_j/\partial w_k(i)$ is nonzero only when $k = j$. Using these results, we find $\partial^+ E/\partial w_k(i)$ to be

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial^+ E}{\partial \mathbf{y}_k} \frac{\partial \mathbf{y}_k}{\partial w_k(i)}. \tag{38}$$

Substituting (38) into (37), the error partial derivative is

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial \mathbf{y}_k} \frac{\partial \mathbf{y}_k}{\partial w_k(i)} = \sum_{k=0}^{k_f} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(i)}, \tag{39}$$

where

$$\lambda_k = \frac{\partial^+ E}{\partial \mathbf{y}_k}$$

is the Lagrange multiplier. Equation (39) gives us Step 3 of the algorithm. The term $\partial \mathbf{y}_k/\partial w_k(i)$ of (39) is easy to calculate. The term $\partial^+ E/\partial \mathbf{y}_k$ can be found using the first chain rule expansion (5) once again.

$$\lambda_k = \frac{\partial^+ E}{\partial \mathbf{y}_k} = \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=k+1}^{k_f} \left(\frac{\partial^+ E}{\partial \mathbf{y}_j} \frac{\partial \mathbf{y}_j}{\partial \mathbf{y}_k} + \frac{\partial^+ E}{\partial \mathbf{W}_j(i)} \frac{\partial \mathbf{W}_j(i)}{\partial \mathbf{y}_k} \right) \tag{40}$$

[4] In our definition of the ordinary partial derivative all terms are held constant except the terms in the denominator of the partial derivative. Therefore, if the function which defines the numerator of the partial derivative does not contain the terms of the denominator directly, then the partial derivative is zero.

All components of the matrix $\partial \mathbf{W}_j(i)/\partial \mathbf{y}_k$ are equal to zero. When $j > k + L$, where L is the maximum number of delays in the system, all components of the matrix $\partial \mathbf{y}_j/\partial \mathbf{y}_k$ are also equal to zero. Using these results, (40) can be written as

$$\lambda_k = \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=1}^{L} \frac{\partial^+ E}{\partial \mathbf{y}_{k+j}} \frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k} \tag{41}$$

$$= \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=1}^{L} \lambda_{k+j} \frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k}. \tag{42}$$

Equation (42) is a backward difference equation that can be solved using the following boundary conditions:

$$\lambda_{k_f} = \frac{\partial E}{\partial \mathbf{y}_{k_f}} \tag{43}$$

$$\lambda_{k_f+1}, \ldots, \lambda_{k_f+L} = 0. \tag{44}$$

Equation (42) gives us Step 2, completing the derivation of Steps 1-3 of the algorithm. As a point of interest, (42), (43) and (44) along with

$$\frac{\partial^+ E}{\partial w(i)} = 0$$

are the discrete-time Euler-Lagrange equations for a dynamical system in the standard representation [7]. The epochwise backsweep algorithm is a numerical technique for solving these equations.

Appendix B: Derivation of the Epochwise Recursive Algorithm

The epochwise backsweep algorithm is derived using the first chain rule expansion, (5). In this section, the second chain rule expansion, (6), is used to derive the epochwise recursive algorithm. Our primary goal in deriving the algorithm is to find an expression for the term $\partial^+ E/\partial w(i)$. We begin by noting that implementation of Step 1-a) of the algorithm along with the calculation of the error results in the set of ordered equations shown in Appendix A (29)–(35). Using the second chain rule expansion, (6), along with this set of ordered equations, the ordered partial derivative of the error may be written as

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left(\frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)} + \frac{\partial E}{\partial \mathbf{W}_k(i)} \frac{\partial^+ \mathbf{W}_k(i)}{\partial w(i)} \right). \tag{45}$$

The term $\partial E/\partial w(i)$ and the components of the vector $\partial E/\partial \mathbf{W}_k(i)$ are equal to zero because the error E is not a direct function of $w(i)$ and $\mathbf{W}_k(i)$. Therefore, (45) can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)}. \tag{46}$$

In the algorithm, this equation is computed recursively using Step 1-c). The first term of (46), $\partial E/\partial \mathbf{y}_k$, is easy to compute. The second term $\partial^+ \mathbf{y}_k/\partial w(i)$ can be found using the second chain rule expansion.

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w(i)} + \sum_{j=0}^{k-1} \frac{\partial \mathbf{y}_k}{\partial \mathbf{W}_j(i)} \frac{\partial^+ \mathbf{W}_j(i)}{\partial w(i)} + \sum_{j=0}^{k-1} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_j} \frac{\partial^+ \mathbf{y}_j}{\partial w(i)} \tag{47}$$

The components of the vector $\partial \mathbf{y}_k/\partial w(i)$ are equal to zero. The term $\partial \mathbf{y}_k/\partial \mathbf{W}_j(i)$ of the first summation is nonzero only when $k = j$, therefore, this summation only contains one nonzero term. As a result, the first summation can be written as $\partial \mathbf{y}_k/\partial w_k(i)$. In addition, the first term of the second summation $\partial \mathbf{y}_k/\partial \mathbf{y}_j$ is nonzero only when $k - j \leq L$. Using these results, (47) can be written as

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \sum_{j=1}^{L} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-j}} \frac{\partial^+ \mathbf{y}_{k-j}}{\partial w(i)}. \tag{48}$$

This equation is used in Step 1-b) to recursively calculate the output gradients for the entire epoch. It is initialized using

$$\frac{\partial^+ \mathbf{y}_{-1}}{\partial w(i)}, \ldots, \frac{\partial^+ \mathbf{y}_{-L}}{\partial w(i)} = 0.$$

Equation (48) completes the derivation of the epochwise recursive algorithm.

Acknowledgments

The author is grateful to Professor Bernard Widrow for his support and encouragement of this research. The author also wants to thank Boyd Fowler and Mike Lehr for their insightful comments on the contents of this paper.

References

[1] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*. Cambridge, MA: The MIT Press, 1986.
[2] R. Hecht-Nielsen. "Theory of the backpropagation neural network," in *Proceedings of International Joint Conference on Neural Networks*, 1989, June, vol. 1, pp. 593–611.
[3] K. Hornik, M. Stinchcombe and H. White, "Multi-layer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 2, pp. 359–366, 1989.
[4] D. G. Luenberger, *Linear and Nonlinear Programming*. Second ed., Reading, MA: Addison-Wesley, 1984.
[5] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, madaline and backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415–1442, 1990.
[6] P. Werbos, "Beyond regression: new tools for prediction and analysis in the behavioral sciences," Ph.D. thesis, Harvard University, Cambridge, MA, August, 1974.
[7] Bryson, Jr. and Y. Ho, *Applied Optimal Control*, New York: Blaisdell Publishing Co., 1969.
[8] K. S. Narendra and L. E. McBride Jr., "Multiparameter self-optimizing system using correlation techniques," *IEEE Trans. on Automat. Contr.*, vol. 9, pp. 31–38, 1964.
[9] L. E. McBride Jr. and K. S. Narendra, "Optimization of time-varying systems," *IEEE Trans. on Automat. Contr.*, vol. 10, pp. 289–294, 1964.
[10] P. V. Kokotovic, "Method of sensitivity points in the investigation and optimization of linear control systems," *Automat. Remote Contr.*, vol. 25, pp. 1512–1518, 1964.
[11] K. S. Narendra and K. Parthasarathy, "Gradient methods for the optimization of dynamic systems containing neural networks," *IEEE Trans. on Neural Networks*, pp. 252–262, 1991.
[12] M. Jordan, "Generic constraints on underspecified target trajectories," in *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, vol. I, June, 1989, pp. 217–225.

[13] D. Nguyen and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," in *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, June, 1989, vol. II, pp. 357–363.
[14] B. Pearlmutter, "Learning state space trajectories in recurrent neural networks," in *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA: Morgan Kauffman, June 17–26, 1988, pp. 113–117.
[15] A. J. Robinson, "Dynamic error propagation networks," Ph.D. thesis, Cambridge University, Cambridge, U.K., June, 1989.
[16] A. J. Robinson and F. Fallside, "The utility driven dynamic error propagation network," Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge University, Cambridge, England, 1987.
[17] P. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, pp. 339–356, 1988.
[18] P. J. Werbos, Learning how the world works: Specifications for predictive networks in robots and brains, *in Proc. 1987 Int. Conf. Syst., Man, Cybern.*, 1987.
[19] F. Beaufays, Y. Abdel-Magid, and B. Widrow, "Application of neural networks to load-frequency control in power systems," *Neural Networks, vol. 7, no. 1, pp. 183–194, 1994.*
[20] S.-Z. Quin, H.-T. Su, and T. J. McAvoy, "Comparison of four neural net learning methods for dynamic system identification," *IEEE Trans. on Neural Networks*, vol. 3, no. 1, pp. 122–130, 1992.
[21] M. I. Jordan and D. E. Rumelhart, "Forward models: Supervised learning with a distal teacher," *Cognitive Science*, July–Sept, vol. 16, no. 3, pp. 307–354, 1992.
[22] M. Kawato, "Computational schemes and neural network models for formation and control of multijoint arm trajectory," in *Neural Networks for Control*, W. T. Miller, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT Press/Bradford Books,1990.
[23] R. Watrous and L. Shastri, "Learning phonetic features using connectionist networks: An experiment in speech recognition," in *Proc. 1st IEEE Int. Conf. on Neural Networks*, June, 1987.
[24] R. Watrous, B. Ladendorf, and G. Kuhn, "Complete gradient optimization of a recurrent network applied to /b/, /d/, /g/ discrimination," *J. Acoust. Soc. Amer.*, vol. 87, pp. 1301–1309, 1990.
[25] H. Sawai, A. Waibel, P. Haffner, M. Miyatake, and K. Shikano, "Parallelism, hierarchy, scaling in time-delay neural networks for spotting Japanese phonemes/CV-syllabales," in *Proc. of the Int. Joint Conf. on Neural Networks*, June, 1989.
[26] M. Gori, Y. Bengio, and R. De Mori, "A learning algorithm for capturing the dynamic nature of speech," in *Proc. of the Int. Joint Conf. on Neural Networks*, vol. II, 1989, pp. 417–423.
[27] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*,pp. 490–501, 1990.
[28] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.
[29] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," ICS Report 8805, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA 92093, October, 1988.
[30] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, pp. 270–277 1989.
[31] M. C. Mozer, "A focused back-propagation algorithm for temporal pattern recognition," Technical Report, Departments of Psycology and Computer Science, University of Toronto, Toronto, 1988.
[32] G. Kuhn, "A first look at phonetic discrimination using a connectionist network with recurrent links," Technical Report, (SCIMP Working Paper No. 4/87), Communications Research Divsion, Insitute for Defense Analysis, Princeton, NJ, 1987.
[33] K. S. Narendra and K. Parthasarathy, "Identification and Control of Dynamic Systems Using Neural Networks," *IEEE Trans. on Neural Networks*, no. 1, pp. 4–27, 1990.
[34] J. Schmidhuber, "An on-line algorithm for dynamic reinforcement learning and planning in reactive environments," in *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, vol. II, pp. 253–257, June, 1990.
[35] G. Kuhn, R. Watrous, and B. Ladendorf, "Connected recognition with a recurrent network," *Speech Communications*, vol. 9, pp. 41-48, 1990.
[36] R. J. Williams, "Adaptive state representation and estimation using recurrent connectionist networks," in *Neural Networks for Control*, W. T. Miller, R. S. Sutto and P. J. Werbos, Eds. Cambridge, MA: MIT Press/Bradford Books, 1990.
[37] F. Beaufays and E. A. Wan, "Relating real-time backpropagation and backpropagation through time: An application of flow graph interreciprocity," Neural Computation, vol. 6, no. 2, 1994.
[38] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1985.
[39] J. J. Shynk, "Adaptive IIR filtering," *IEEE ASSP Magazine*, April, 1989.
[40] G. Franklin and J. Powell, *Digital Control of Dynamic Systems*. Reading, MA: Addison-Wesley, 1980.
[41] T. Kailath, *Linear Systems*, Englewood Cliffs, NJ: Prentice Hall, 1980.
[42] D. Psaltis, A. Sideris, and A. A. Yamamura, "A multilayered neural network controller," *IEEE Control Magazine*, pp. 17–21, 1988.
[43] D. Nguyen and B. Widrow, "Neural networks for self-learning control systems," *IEEE Control Systems Magazine*, April, 1990.
[44] Y. le Cun, "A theoretical framework for back-propagation," in *Proceedings of the 1988 Connectionist Models Summer School*, June 17-26, San Mateo, CA: Morgan Kauffman, 1988, pp. 21–28.
[45] D. B. Parker, "Learning-logic," Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University, Stanford, CA, October, 1982.
[46] M. Gherrity, "A learning algorithm for analog, fully recurrent neural networks," in *Proceedings of the International Joint Conference on Neural Networks*, vol. I, Washington, DC, June, 1989, pp. 643–644.
[47] J. L. Elman, "Finding structure in time," *Cognitive Science*, pp. 179–211, 1990.
[48] S. A. White, "An adaptive recursive digital filter," in *Proc. 9th Asilomar Conf. Circuits Syst. Compt.*, Nov., 1975, pp. 21.

Stephen Piché, (S'90–M'92–M'92), received the B.S. degree in electrical engineering from the University of Colorado, Boulder, and the M.S., and Ph.D. degrees in electrical engineering from Stanford University in 1989 and 1992 respectively.

He is currently a member of the Neural Network Project at the Microelectronics and Computer Technology Corporation (MCC) in Austin, Texas. His research interests include neural network theory, and the application of connectionist models in the areas of control, signal processing, and finance. He is a member of the IEEE and INNS.